
Table of Contents

Introduction	1.1
Spark SQL — Structured Data Processing with Relational Queries on Massive Scale	
Datasets vs DataFrames vs RDDs	1.3 1.2
Dataset API vs SQL	1.4

(WIP) Vectorized Parquet Decoding

VectorizedParquetRecordReader	2.1
VectorizedColumnReader	2.2
SpecificParquetRecordReaderBase	2.3
ColumnVector Contract — In-Memory Columnar Data	2.4
WritableColumnVector Contract	2.4.1
OnHeapColumnVector	2.4.2
OffHeapColumnVector	2.4.3

Notable Features

Vectorized Parquet Decoding (Reader)	3.1
Dynamic Partition Inserts	3.2
Bucketing	3.3
Whole-Stage Java Code Generation (Whole-Stage CodeGen)	3.4
CodegenContext	3.4.1
CodeGenerator	3.4.2
GenerateColumnAccessor	3.4.2.1
GenerateOrdering	3.4.2.2
GeneratePredicate	3.4.2.3
GenerateSafeProjection	3.4.2.4
BytesToBytesMap Append-Only Hash Map	3.4.3
Vectorized Query Execution (Batch Decoding)	3.5
ColumnarBatch — ColumnVectors as Row-Wise Table	3.5.1

Data Source API V2	3.6
Subqueries	3.7
Hint Framework	3.8
Adaptive Query Execution	3.9
ExchangeCoordinator	3.9.1
Subexpression Elimination For Code-Generated Expression Evaluation (Common Expression Reuse)	3.10
EquivalentExpressions	3.10.1
Cost-Based Optimization (CBO)	3.11
CatalogStatistics — Table Statistics in Metastore (External Catalog)	3.11.1
ColumnStat — Column Statistics	3.11.2
EstimationUtils	3.11.3
CommandUtils — Utilities for Table Statistics	3.11.4
Catalyst DSL — Implicit Conversions for Catalyst Data Structures	3.12

Developing Spark SQL Applications

Fundamentals of Spark SQL Application Development	4.1
SparkSession — The Entry Point to Spark SQL	4.2
Builder — Building SparkSession using Fluent API	4.2.1
implicits Object — Implicits Conversions	4.2.2
SparkSessionExtensions	4.2.3
Dataset — Structured Query with Data Encoder	4.3
DataFrame — Dataset of Rows with RowEncoder	4.3.1
Row	4.3.2
DataSource API — Managing Datasets in External Data Sources	4.4
DataFrameReader — Loading Data From External Data Sources	4.4.1
DataFrameWriter — Saving Data To External Data Sources	4.4.2
Dataset API — Dataset Operators	4.5
Typed Transformations	4.5.1
Untyped Transformations	4.5.2
Basic Actions	4.5.3
Actions	4.5.4
DataFrameNaFunctions — Working With Missing Data	4.5.5

DataFrameStatFunctions — Working With Statistic Functions	4.5.6
Column	4.6
Column API — Column Operators	4.6.1
TypedColumn	4.6.2
Basic Aggregation — Typed and Untyped Grouping Operators	4.7
RelationalGroupedDataset — Untyped Row-based Grouping	4.7.1
KeyValueGroupedDataset — Typed Grouping	4.7.2
Dataset Join Operators	4.8
Broadcast Joins (aka Map-Side Joins)	4.8.1
Window Aggregation	4.9
WindowSpec — Window Specification	4.9.1
Window Utility Object — Defining Window Specification	4.9.2
Standard Functions — functions Object	4.10
Aggregate Functions	4.10.1
Collection Functions	4.10.2
Date and Time Functions	4.10.3
Regular Functions (Non-Aggregate Functions)	4.10.4
Window Aggregation Functions	4.10.5
User-Defined Functions (UDFs)	4.11
UDFs are Blackbox — Don't Use Them Unless You've Got No Choice	4.11.1
UserDefinedFunction	4.11.2
Schema — Structure of Data	4.12
StructType	4.12.1
StructField — Single Field in StructType	4.12.2
Data Types	4.12.3
Multi-Dimensional Aggregation	4.13
Dataset Caching and Persistence	4.14
User-Friendly Names Of Cached Queries in web UI's Storage Tab	4.14.1
Dataset Checkpointing	4.15
UserDefinedAggregateFunction — Contract for User-Defined Untyped Aggregate Functions (UDAFs)	4.16
Aggregator — Contract for User-Defined Typed Aggregate Functions (UDAFs)	4.17
Configuration Properties	4.18

SparkSession Registries

Catalog — Metastore Management Interface	5.1
CatalogImpl	5.1.1
ExecutionListenerManager — Management Interface of QueryExecutionListeners	5.2
ExperimentalMethods	5.3
ExternalCatalog Contract — External Catalog (Metastore) of Permanent Relational Entities	5.4
InMemoryCatalog	5.4.1
HiveExternalCatalog — Hive-Aware Metastore of Permanent Relational Entities	
FunctionRegistry — Contract for Function Registries (Catalogs)	5.5 5.4.2
GlobalTempViewManager — Management Interface of Global Temporary Views	5.6
SessionCatalog — Session-Scoped Catalog of Relational Entities	5.7
CatalogTable — Table Specification (Native Table Metadata)	5.7.1
CatalogStorageFormat — Storage Specification of Table or Partition	5.7.1.1
CatalogTablePartition — Partition Specification of Table	5.7.1.2
BucketSpec — Bucketing Specification of Table	5.7.1.3
HiveSessionCatalog — Hive-Specific Catalog of Relational Entities	5.7.2
HiveMetastoreCatalog — Legacy SessionCatalog for Converting Hive Metastore Relations to Data Source Relations	5.7.3
SessionState	5.8
BaseSessionStateBuilder — Generic Builder of SessionState	5.8.1
SessionStateBuilder	5.8.2
HiveSessionStateBuilder — Builder of Hive-Specific SessionState	5.8.3
SharedState — State Shared Across SparkSessions	5.9
CacheManager — In-Memory Cache for Tables and Views	5.10
CachedRDDBuilder	5.10.1
RuntimeConfig — Management Interface of Runtime Configuration	5.11
SQLConf — Internal Configuration Store	5.12
StaticSQLConf — Cross-Session, Immutable and Static SQL Configuration	5.12.1
CatalystConf	5.12.2
UDFRegistration — Session-Scoped FunctionRegistry	5.13

File-Based Data Sources

FileFormat	6.1
OrcFileFormat	6.1.1
ParquetFileFormat	6.1.2
TextBasedFileFormat	6.2
CSVFileFormat	6.2.1
JsonFileFormat	6.2.2
TextFileFormat	6.2.3
JsonDataSource	6.3
FileCommitProtocol	6.4
SQLHadoopMapReduceCommitProtocol	6.4.1
PartitionedFile — File Block in FileFormat Data Source	6.5
FileScanRDD — Input RDD of FileSourceScanExec Physical Operator	6.6
ParquetReadSupport — Non-Vectorized ReadSupport in Parquet Data Source	6.7
RecordReaderIterator — Scala Iterator over Hadoop RecordReader's Values	6.8

Kafka Data Source

Kafka Data Source	7.1
Kafka Data Source Options	7.2
KafkaSourceProvider	7.3
KafkaRelation	7.4
KafkaSourceRDD	7.5
KafkaSourceRDDOffsetRange	7.5.1
KafkaSourceRDDPartition	7.5.2
ConsumerStrategy Contract — Kafka Consumer Providers	7.6
KafkaOffsetReader	7.7
KafkaOffsetRangeLimit	7.8
KafkaDataConsumer Contract	7.9
InternalKafkaConsumer	7.9.1
KafkaWriter Helper Object — Writing Structured Queries to Kafka	7.10
KafkaWriteTask	7.10.1
JsonUtils Helper Object	7.11

Avro Data Source

Avro Data Source	8.1
AvroFileFormat — FileFormat For Avro-Encoded Files	8.2
AvroOptions — Avro Data Source Options	8.3
CatalystDataToAvro Unary Expression	8.4
AvroDataToCatalyst Unary Expression	8.5

JDBC Data Source

JDBC Data Source	9.1
JDBCOPTIONS	9.2
JdbcRelationProvider	9.3
JDBCRelation	9.4
JDBCRDD	9.5
JdbcDialect	9.6
JdbcUtils Helper Object	9.7

Console Data Source

Console Data Source	10.1
ConsoleSinkProvider	10.2

Hive Data Source

Hive Integration	11.1
Hive Metastore	11.1.1
Spark SQL CLI — spark-sql	11.1.2
DataSinks Strategy	11.1.3
HiveFileFormat	11.2
HiveClient	11.3
HiveClientImpl — The One and Only HiveClient	11.4
HiveUtils	11.5

Extending Spark SQL / Data Source API V2

DataSourceV2 — Data Sources in Data Source API V2	12.1
ReadSupport Contract — "Readable" Data Sources	12.2
WriteSupport Contract — "Writable" Data Sources	12.3
DataSourceReader	12.4
SupportsPushDownFilters	12.4.1
SupportsPushDownRequiredColumns	12.4.2
SupportsReportPartitioning	12.4.3
SupportsReportStatistics	12.4.4
SupportsScanColumnarBatch	12.4.5
DataSourceWriter	12.5
SessionConfigSupport	12.6
InputPartition	12.7
InputPartitionReader	12.8
DataWriter	12.9
DataWriterFactory	12.10
InternalRowDataWriterFactory	12.10.1
DataSourceV2StringFormat	12.11
DataSourceRDD — Input RDD Of DataSourceV2ScanExec Physical Operator	12.12
DataSourceRDDPartition	12.12.1
DataWritingSparkTask Partition Processing Function	12.13
DataSourceV2Utils Helper Object	12.14

Extending Spark SQL / Data Source API V1

DataSource — Pluggable Data Provider Framework	13.1
Custom Data Source Formats	13.2

Data Source Providers

CreatableRelationProvider Contract — Data Sources That Write Rows Per Save Mode	
DataSourceRegister Contract — Registering Data Source Format	14.2
RelationProvider Contract — Relation Providers With Schema Inference	14.3

SchemaRelationProvider Contract — Relation Providers With Mandatory User-Defined Schema	14.4
---	------

Data Source Relations / Extension Contracts

BaseRelation — Collection of Tuples with Schema	15.1
HadoopFsRelation — Relation for File-Based Data Source	15.1.1
CatalystScan Contract	15.2
InsertableRelation Contract — Non-File-Based Relations with Inserting or Overwriting Data Support	15.3
PrunedFilteredScan Contract — Relations with Column Pruning and Filter Pushdown	
PrunedScan Contract	15.4
TableScan Contract — Relations with Column Pruning	15.5
	15.6

Others

FileFormatWriter Helper Object	16.1
Data Source Filter Predicate (For Filter Pushdown)	16.2
FileRelation Contract	16.3

Structured Query Execution

QueryExecution — Structured Query Execution Pipeline	17.1
UnsupportedOperationChecker	17.1.1
Analyzer — Logical Query Plan Analyzer	17.2
CheckAnalysis — Analysis Validation	17.2.1
SparkOptimizer — Logical Query Plan Optimizer	17.3
Catalyst Optimizer — Generic Logical Query Plan Optimizer	17.3.1
SparkPlanner — Spark Query Planner	17.4
SparkStrategy — Base for Execution Planning Strategies	17.4.1
SparkStrategies — Container of Execution Planning Strategies	17.4.2
LogicalPlanStats — Statistics Estimates and Query Hints of Logical Operator	17.5
Statistics — Estimates of Plan Statistics and Query Hints	17.5.1
HintInfo	17.5.2

LogicalPlanVisitor — Base Visitor for Computing Statistics of Logical Plan	17.5.3
SizeInBytesOnlyStatsPlanVisitor — LogicalPlanVisitor for Total Size (in Bytes) Statistic Only	17.5.4
BasicStatsPlanVisitor — Computing Statistics for Cost-Based Optimization	17.5.5
AggregateEstimation	17.5.5.1
FilterEstimation	17.5.5.2
JoinEstimation	17.5.5.3
ProjectEstimation	17.5.5.4
Partitioning — Specification of Physical Operator's Output Partitions	17.6
Distribution Contract — Data Distribution Across Partitions	17.7
AllTuples	17.7.1
BroadcastDistribution	17.7.2
ClusteredDistribution	17.7.3
HashClusteredDistribution	17.7.4
OrderedDistribution	17.7.5
UnspecifiedDistribution	17.7.6

Catalyst Expressions

Catalyst Expression — Executable Node in Catalyst Tree	18.1
AggregateExpression	18.2
AggregateFunction Contract — Aggregate Function Expressions	18.3
AggregateWindowFunction Contract — Declarative Window Aggregate Function Expressions	18.4
AttributeReference	18.5
Alias	18.6
Attribute	18.7
BoundReference	18.8
CallMethodViaReflection	18.9
Coalesce	18.10
CodegenFallback	18.11
CollectionGenerator	18.12
ComplexTypedAggregateExpression	18.13
CreateArray	18.14

CreateNamedStruct	18.15
CreateNamedStructLike Contract	18.16
CreateNamedStructUnsafe	18.17
CumeDist	18.18
DeclarativeAggregate Contract — Unevaluable Aggregate Function Expressions	18.19
ExecSubqueryExpression	18.20
Exists	18.21
ExpectsInputTypes Contract	18.22
ExplodeBase Contract	18.23
First	18.24
Generator	18.25
GetArrayStructFields	18.26
GetArrayItem	18.27
GetMapValue	18.28
GetStructField	18.29
ImperativeAggregate	18.30
In	18.31
Inline	18.32
InSet	18.33
InSubquery	18.34
JsonToStructs	18.35
JsonTuple	18.36
ListQuery	18.37
Literal	18.38
MonotonicallyIncreasingID	18.39
Murmur3Hash	18.40
NamedExpression Contract	18.41
Nondeterministic Contract	18.42
OffsetWindowFunction Contract — Unevaluable Window Function Expressions	18.43
ParseToDate	18.44
ParseToTimestamp	18.45
PlanExpression	18.46
PrettyAttribute	18.47
RankLike Contract	18.48

ResolvedStar	18.49
RowNumberLike Contract	18.50
RuntimeReplaceable Contract	18.51
ScalarSubquery SubqueryExpression	18.52
ScalarSubquery ExecSubqueryExpression	18.53
ScalaUDF	18.54
ScalaUDAF	18.55
SimpleTypedAggregateExpression	18.56
SizeBasedWindowFunction Contract — Declarative Window Aggregate Functions with Window Size	18.57
SortOrder	18.58
Stack	18.59
Star	18.60
StaticInvoke	18.61
SubqueryExpression	18.62
TimeWindow	18.63
TypedAggregateExpression	18.64
TypedImperativeAggregate	18.65
UnaryExpression Contract	18.66
UnixTimestamp	18.67
UnresolvedAttribute	18.68
UnresolvedFunction	18.69
UnresolvedGenerator	18.70
UnresolvedOrdinal	18.71
UnresolvedRegex	18.72
UnresolvedStar	18.73
UnresolvedWindowExpression	18.74
WindowExpression	18.75
WindowFunction Contract — Window Function Expressions With WindowFrame	18.76
WindowSpecDefinition	18.77

Logical Operators

Base Logical Operators (Contracts)

LogicalPlan Contract — Logical Operator with Children and Expressions / Logical Query Plan	20.1
Command Contract — Eagerly-Executed Logical Operator	20.2
RunnableCommand Contract — Generic Logical Command with Side Effects	20.3
DataWritingCommand Contract — Logical Commands That Write Query Data	20.4
SaveAsHiveFile Contract — DataWritingCommands That Write Query Result As Hive Files	20.5

Concrete Logical Operators

Aggregate	21.1
AlterViewAsCommand	21.2
AnalysisBarrier	21.3
AnalyzeColumnCommand	21.4
AnalyzePartitionCommand	21.5
AnalyzeTableCommand	21.6
AppendData	21.7
ClearCacheCommand	21.8
CreateDataSourceTableAsSelectCommand	21.9
CreateDataSourceTableCommand	21.10
CreateHiveTableAsSelectCommand	21.11
CreateTable	21.12
CreateTableCommand	21.13
CreateTempViewUsing	21.14
CreateViewCommand	21.15
DataSourceV2Relation	21.16
DescribeColumnCommand	21.17
DescribeTableCommand	21.18
DeserializeToObject	21.19
DropTableCommand	21.20
Except	21.21
Expand	21.22

ExplainCommand	21.23
ExternalRDD	21.24
Filter	21.25
Generate	21.26
GroupingSets	21.27
Hint	21.28
HiveTableRelation	21.29
InMemoryRelation	21.30
InsertIntoDataSourceCommand	21.31
InsertIntoDataSourceDirCommand	21.32
InsertIntoDir	21.33
InsertIntoHadoopFsRelationCommand	21.34
InsertIntoHiveDirCommand	21.35
InsertIntoHiveTable	21.36
InsertIntoTable	21.37
Intersect	21.38
Join	21.39
LeafNode	21.40
LocalRelation	21.41
LogicalRDD	21.42
LogicalRelation	21.43
OneRowRelation	21.44
Pivot	21.45
Project	21.46
Range	21.47
Repartition and RepartitionByExpression	21.48
ResolvedHint	21.49
SaveIntoDataSourceCommand	21.50
ShowCreateTableCommand	21.51
ShowTablesCommand	21.52
Sort	21.53
SubqueryAlias	21.54
TypedFilter	21.55
Union	21.56

UnresolvedCatalogRelation	21.57
UnresolvedHint	21.58
UnresolvedInlineTable	21.59
UnresolvedRelation	21.60
UnresolvedTableValuedFunction	21.61
Window	21.62
WithWindowDefinition	21.63
WriteToDataSourceV2	21.64
View	21.65

Physical Operators

SparkPlan Contract — Physical Operators in Physical Query Plan of Structured Query	
CodegenSupport Contract — Physical Operators with Java Code Generation	22.1
DataSourceScanExec Contract — Leaf Physical Operators to Scan Over	22.2
BaseRelation	22.3
ColumnarBatchScan Contract — Physical Operators With Vectorized Reader	22.4
ObjectConsumerExec Contract — Unary Physical Operators with Child Physical Operator with One-Attribute Output Schema	22.5
BaseLimitExec Contract	22.6
Exchange Contract	22.7
Projection Contract — Functions to Produce InternalRow for InternalRow	22.8
UnsafeProjection — Generic Function to Project InternalRows to UnsafeRows	
GenerateUnsafeProjection	22.8.2 22.8.1
GenerateMutableProjection	22.8.3
InterpretedProjection	22.8.4
CodeGeneratorWithInterpretedFallback	22.8.5
SQLMetric — SQL Execution Metric of Physical Operator	22.9

Concrete Physical Operators

BroadcastExchangeExec	23.1
BroadcastHashJoinExec	23.2
BroadcastNestedLoopJoinExec	23.3

CartesianProductExec	23.4
CoalesceExec	23.5
DataSourceV2ScanExec	23.6
DataWritingCommandExec	23.7
DebugExec	23.8
DeserializeToObjectExec	23.9
ExecutedCommandExec	23.10
ExpandExec	23.11
ExternalRDDScanExec	23.12
FileSourceScanExec	23.13
FilterExec	23.14
GenerateExec	23.15
HashAggregateExec	23.16
HiveTableScanExec	23.17
InMemoryTableScanExec	23.18
LocalTableScanExec	23.19
MapElementsExec	23.20
ObjectHashAggregateExec	23.21
ObjectProducerExec	23.22
ProjectExec	23.23
RangeExec	23.24
RDDScanExec	23.25
ReusedExchangeExec	23.26
RowDataSourceScanExec	23.27
SampleExec	23.28
ShuffleExchangeExec	23.29
ShuffledHashJoinExec	23.30
SerializeFromObjectExec	23.31
SortAggregateExec	23.32
SortMergeJoinExec	23.33
SortExec	23.34
SubqueryExec	23.35
InputAdapter	23.36
WindowExec	23.37

AggregateProcessor	23.37.1
WindowFunctionFrame	23.37.2
WholeStageCodegenExec	23.38
WriteToDataSourceV2Exec	23.39

Logical Analysis Rules (Check, Evaluation, Conversion and Resolution)

AliasViewChild	24.1
CleanupAliases	24.2
DataSourceAnalysis	24.3
DetermineTableStats	24.4
ExtractWindowExpressions	24.5
FindDataSourceTable	24.6
HandleNullInputsForUDF	24.7
HiveAnalysis	24.8
InConversion	24.9
LookupFunctions	24.10
PreprocessTableCreation	24.11
PreWriteCheck	24.12
RelationConversions	24.13
ResolveAliases	24.14
ResolveBroadcastHints	24.15
ResolveCoalesceHints	24.16
ResolveCreateNamedStruct	24.17
ResolveFunctions	24.18
ResolveHiveSerdeTable	24.19
ResolveInlineTables	24.20
ResolveMissingReferences	24.21
ResolveOrdinalInOrderByAndGroupBy	24.22
ResolveOutputRelation	24.23
ResolveReferences	24.24
ResolveRelations	24.25

ResolveSQLOnFile	24.26
ResolveSubquery	24.27
ResolveWindowFrame	24.28
ResolveWindowOrder	24.29
TimeWindowing	24.30
UpdateOuterReferences	24.31
WindowFrameCoercion	24.32
WindowsSubstitution	24.33

Base Logical Optimizations (Optimizer)

CollapseWindow	25.1
ColumnPruning	25.2
CombineTypedFilters	25.3
CombineUnions	25.4
ComputeCurrentTime	25.5
ConstantFolding	25.6
CostBasedJoinReorder	25.7
DecimalAggregates	25.8
EliminateSerialization	25.9
EliminateSubqueryAliases	25.10
EliminateView	25.11
GetCurrentDatabase	25.12
LimitPushDown	25.13
NullPropagation	25.14
Optimizeln	25.15
OptimizeSubqueries	25.16
PropagateEmptyRelation	25.17
PullupCorrelatedPredicates	25.18
PushDownPredicate	25.19
PushPredicateThroughJoin	25.20
ReorderJoin	25.21
ReplaceExpressions	25.22
RewriteCorrelatedScalarSubquery	25.23

RewritePredicateSubquery	25.24
SimplifyCasts	25.25

Extended Logical Optimizations (SparkOptimizer)

ExtractPythonUDFFromAggregate	26.1
OptimizeMetadataOnlyQuery	26.2
PruneFileSourcePartitions	26.3
PushDownOperatorsToDataSource	26.4

Execution Planning Strategies

Aggregation	27.1
BasicOperators	27.2
DataSourceStrategy	27.3
DataSourceV2Strategy	27.4
FileSourceStrategy	27.5
HiveTableScans	27.6
InMemoryScans	27.7
JoinSelection	27.8
SpecialLimits	27.9

Physical Query Optimizations

CollapseCodegenStages	28.1
EnsureRequirements	28.2
ExtractPythonUDFs	28.3
PlanSubqueries	28.4
ReuseExchange	28.5
ReuseSubquery	28.6

Encoders

Encoder — Internal Row Converter	29.1
Encoders Factory Object	29.1.1
ExpressionEncoder — Expression-Based Encoder	29.1.2
RowEncoder — Encoder for DataFrames	29.1.3
LocalDateTimeEncoder — Custom ExpressionEncoder for java.time.LocalDateTime	29.1.4

RDDs

ShuffledRowRDD	30.1
----------------	------

Monitoring

SQL Tab — Monitoring Structured Queries in web UI	31.1
SQLListener Spark Listener	31.1.1
QueryExecutionListener	31.2
SQLAppStatusListener Spark Listener	31.3
SQLAppStatusPlugin	31.4
SQLAppStatusStore	31.5
WriteTaskStats	31.6
BasicWriteTaskStats	31.6.1
WriteTaskStatsTracker	31.7
BasicWriteTaskStatsTracker	31.7.1
WriteJobStatsTracker	31.8
BasicWriteJobStatsTracker	31.8.1
Logging	31.9

Performance Tuning and Debugging

Spark SQL's Performance Tuning Tips and Tricks (aka Case Studies)	32.1
Number of Partitions for groupBy Aggregation	32.1.1
Debugging Query Execution	32.2

Catalyst—Tree Manipulation Framework

Catalyst—Tree Manipulation Framework	33.1
TreeNode—Node in Catalyst Tree	33.2
QueryPlan—Structured Query Plan	33.2.1
RuleExecutorContract—Tree Transformation Rule Executor	33.3
Catalyst Rule—Named Transformation of TreeNodes	33.3.1
QueryPlanner—Converting Logical Plan to Physical Trees	33.4
GenericStrategy	33.5

Tungsten Execution Backend

Tungsten Execution Backend (Project Tungsten)	34.1
InternalRow—Abstract Binary Row Format	34.2
UnsafeRow—Mutable Raw-Memory Unsafe Binary Row Format	34.2.1
AggregationIterator—Generic Iterator of UnsafeRows for Aggregate Physical Operators	34.3
ObjectAggregationIterator	34.3.1
SortBasedAggregationIterator	34.3.2
TungstenAggregationIterator—Iterator of UnsafeRows for HashAggregateExec Physical Operator	34.3.3
CatalystSerde	34.4
ExternalAppendOnlyUnsafeRowArray—Append-Only Array for UnsafeRows (with Disk Spill Threshold)	34.5
UnsafeFixedWidthAggregationMap	34.6

SQL Support

SQL Parsing Framework	35.1
AbstractSqlParser	35.2
AstBuilder	35.3
CatalystSqlParser	35.4
ParserInterface	35.5
SparkSqlAstBuilder	35.6

Spark Thrift Server

Thrift JDBC/ODBC Server — Spark Thrift Server (STS)	36.1
SparkSQLEnv	36.2

Varia / Uncategorized

SQLExecution Helper Object	37.1
RDDConversions Helper Object	37.2
CatalystTypeConverters Helper Object	37.3
StatFunctions Helper Object	37.4
SubExprUtils Helper Object	37.5
PredicateHelper Scala Trait	37.6
SchemaUtils Helper Object	37.7
AggUtils Helper Object	37.8
ScalaReflection	37.9
CreateStruct Function Builder	37.10
MultiInstanceRelation	37.11
TypeCoercion Object	37.12
TypeCoercionRule — Contract For Type Coercion Rules	37.13
ExtractEquiJoinKeys — Scala Extractor for Destructuring Join Logical Operators	37.14
PhysicalAggregation — Scala Extractor for Destructuring Aggregate Logical Operators	
PhysicalOperation — Scala Extractor for Destructuring Logical Query Plans	37.15
HashJoin — Contract for Hash-based Join Physical Operators	37.17 37.16
HashedRelation	37.18
LongHashedRelation	37.18.1
UnsafeHashedRelation	37.18.2
KnownSizeEstimation	37.19
SizeEstimator	37.20
BroadcastMode	37.21
HashedRelationBroadcastMode	37.21.1
IdentityBroadcastMode	37.21.2

PartitioningUtils	37.22
HadoopFileLinesReader	37.23
CatalogUtils Helper Object	37.24
ExternalCatalogUtils	37.25
PartitioningAwareFileIndex	37.26
BufferedRowIterator	37.27
CompressionCodecs	37.28
(obsolete) SQLContext	37.29

The Internals of Spark SQL (Apache Spark 2.4.4)

Welcome to **The Internals of Spark SQL** gitbook! I'm very excited to have you here and hope you will enjoy exploring the internals of Spark SQL as much as I have.

I write to discover what I know.

— Flannery O'Connor

I'm [Jacek Laskowski](#), a freelance IT consultant, software engineer and technical instructor specializing in [Apache Spark](#), [Apache Kafka](#) and [Kafka Streams](#) (with [Scala](#) and [sbt](#)).

I offer software development and consultancy services with hands-on in-depth workshops and mentoring. Reach out to me at jacek@japila.pl or [@jaceklaskowski](https://twitter.com/jaceklaskowski) to discuss opportunities.

Consider joining me at [Warsaw Scala Enthusiasts](#) and [Warsaw Spark](#) meetups in Warsaw, Poland.

Tip

I'm also writing other books in the "The Internals of" series about [Apache Spark](#), [Spark Structured Streaming](#), [Apache Kafka](#), and [Kafka Streams](#).

Expect text and code snippets from a variety of public sources. Attribution follows.

Now, let me introduce you to [Spark SQL and Structured Queries](#).

Spark SQL — Structured Data Processing with Relational Queries on Massive Scale

Like Apache Spark in general, **Spark SQL** in particular is all about distributed in-memory computations on massive scale.

Quoting the [Spark SQL: Relational Data Processing in Spark](#) paper on Spark SQL:

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API.

Spark SQL lets Spark programmers leverage the benefits of relational processing (e.g., declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (e.g., machine learning).

The primary difference between the computation models of Spark SQL and Spark Core is the relational framework for ingesting, querying and persisting (semi)structured data using **relational queries** (aka **structured queries**) that can be expressed in *good ol' SQL* (with many features of HiveQL) and the high-level SQL-like functional declarative [Dataset API](#) (aka **Structured Query DSL**).

Note

Semi- and structured data are collections of records that can be described using [schema](#) with column names, their types and whether a column can be null or not (aka *nullability*).

Whichever query interface you use to describe a structured query, i.e. SQL or Query DSL, the query becomes a [Dataset](#) (with a mandatory [Encoder](#)).

From [Shark, Spark SQL, Hive on Spark, and the future of SQL on Apache Spark](#):

For **SQL users**, Spark SQL provides state-of-the-art SQL performance and maintains compatibility with Shark/Hive. In particular, like Shark, Spark SQL supports all existing Hive data formats, user-defined functions (UDF), and the Hive metastore.

For **Spark users**, Spark SQL becomes the narrow-waist for manipulating (semi-) structured data as well as ingesting data from sources that provide schema, such as JSON, Parquet, Hive, or EDWs. It truly unifies SQL and sophisticated analysis, allowing users to mix and match SQL and more imperative programming APIs for advanced analytics.

For **open source hackers**, Spark SQL proposes a novel, elegant way of building query planners. It is incredibly easy to add new optimizations under this framework.

A `Dataset` is a programming interface to the [structured query execution pipeline](#) with [transformations and actions](#) (as in the good old days of RDD API in Spark Core).

Internally, a structured query is a [Catalyst tree](#) of (logical and physical) [relational operators](#) and [expressions](#).

When an action is executed on a `Dataset` (directly, e.g. [show](#) or [count](#), or indirectly, e.g. [save](#) or [saveAsTable](#)) the structured query (behind `Dataset`) goes through the [execution stages](#):

1. Logical Analysis
2. Caching Replacement
3. Logical Query Optimization (using [rule-based](#) and [cost-based](#) optimizations)
4. Physical Planning
5. Physical Optimization (e.g. [Whole-Stage Java Code Generation](#) or [Adaptive Query Execution](#))
6. Constructing the RDD of Internal Binary Rows (that represents the structured query in terms of Spark Core's RDD API)

As of Spark 2.0, Spark SQL is now *de facto* the primary and feature-rich interface to Spark's underlying in-memory distributed platform (hiding Spark Core's RDDs behind higher-level abstractions that allow for [logical](#) and [physical](#) query optimization strategies even without your consent).

Note	You can find out more on the core of Apache Spark (aka <i>Spark Core</i>) in Mastering Apache Spark 2 gitbook.
------	---

In other words, Spark SQL's `Dataset` API describes a distributed computation that will eventually be converted to a [DAG of RDDs](#) for execution.

Note	Under the covers, structured queries are automatically compiled into corresponding RDD operations.
------	--

Spark SQL supports structured queries in **batch** and **streaming** modes (with the latter as a separate module of Spark SQL called **Spark Structured Streaming**).

Note	You can find out more on Spark Structured Streaming in Spark Structured Streaming (Apache Spark 2.2+) gitbook.
------	--

```
// Define the schema using a case class
case class Person(name: String, age: Int)

// you could read people from a CSV file
// It's been a while since you saw RDDs, hasn't it?
// Excuse me for bringing you the old past.
import org.apache.spark.rdd.RDD
val peopleRDD: RDD[Person] = sc.parallelize(Seq(Person("Jacek", 10)))

// Convert RDD[Person] to Dataset[Person] and run a query

// Automatic schema inference from existing RDDs
scala> val people = peopleRDD.toDS
people: org.apache.spark.sql.Dataset[Person] = [name: string, age: int]

// Query for teenagers using Scala Query DSL
scala> val teenagers = people.where('age >= 10).where('age <= 19).select('name).as[String]
teenagers: org.apache.spark.sql.Dataset[String] = [name: string]

scala> teenagers.show
+---+
| name|
+---+
| Jacek|
+---+

// You could however want to use good ol' SQL, couldn't you?

// 1. Register people Dataset as a temporary view in Catalog
people.createOrReplaceTempView("people")

// 2. Run SQL query
val teenagers = sql("SELECT * FROM people WHERE age >= 10 AND age <= 19")
scala> teenagers.show
+---+---+
| name|age|
+---+---+
| Jacek| 10|
+---+---+
```

Spark SQL supports loading datasets from various data sources including tables in Apache Hive. With Hive support enabled, you can load datasets from existing Apache Hive deployments and save them back to Hive tables if needed.

```

sql("CREATE OR REPLACE TEMPORARY VIEW v1 (key INT, value STRING) USING csv OPTIONS ('path'='people.csv', 'header'='true')")

// Queries are expressed in HiveQL
sql("FROM v1").show

scala> sql("desc EXTENDED v1").show(false)
+-----+-----+-----+
|col_name|data_type|comment|
+-----+-----+-----+
|# col_name|data_type|comment|
|key      |int       |null    |
|value    |string    |null    |
+-----+-----+-----+

```

Like SQL and NoSQL databases, Spark SQL offers performance query optimizations using [rule-based query optimizer](#) (aka **Catalyst Optimizer**), [whole-stage Java code generation](#) (aka **Whole-Stage Codegen** that could often be better than your own custom hand-written code!) and [Tungsten execution engine](#) with its own [internal binary row format](#).

As of Spark SQL 2.2, structured queries can be further optimized using [Hint Framework](#).

Spark SQL introduces a tabular data abstraction called [Dataset](#) (that was previously [DataFrame](#)). `Dataset` data abstraction is designed to make processing large amount of structured tabular data on Spark infrastructure simpler and faster.

Note

Quoting [Apache Drill](#) which applies to Spark SQL perfectly:

A SQL query engine for relational and NoSQL databases with direct queries on self-describing and semi-structured data in files, e.g. JSON or Parquet, and HBase tables without needing to specify metadata definitions in a centralized store.

The following snippet shows a **batch ETL pipeline** to process JSON files and saving their subset as CSVs.

```

spark.read
  .format("json")
  .load("input-json")
  .select("name", "score")
  .where($"score" > 15)
  .write
  .format("csv")
  .save("output-csv")

```

With [Structured Streaming](#) feature however, the above static batch query becomes dynamic and continuous paving the way for **continuous applications**.

```

import org.apache.spark.sql.types._
val schema = StructType(
  StructField("id", LongType, nullable = false) ::
  StructField("name", StringType, nullable = false) ::
  StructField("score", DoubleType, nullable = false) :: Nil)

spark.readStream
  .format("json")
  .schema(schema)
  .load("input-json")
  .select("name", "score")
  .where('score > 15)
  .writeStream
  .format("console")
  .start

// -----
// Batch: 1
// -----
// +---+---+
// | name|score|
// +---+---+
// | Jacek| 20.5|
// +---+---+

```

As of Spark 2.0, the main data abstraction of Spark SQL is [Dataset](#). It represents a **structured data** which are records with a known schema. This structured data representation `Dataset` enables [compact binary representation](#) using compressed columnar format that is stored in managed objects outside JVM's heap. It is supposed to speed computations up by reducing memory usage and GCs.

Spark SQL supports [predicate pushdown](#) to optimize performance of Dataset queries and can also [generate optimized code at runtime](#).

Spark SQL comes with the different APIs to work with:

1. [Dataset API](#) (formerly [DataFrame API](#)) with a strongly-typed LINQ-like Query DSL that Scala programmers will likely find very appealing to use.
2. [Structured Streaming API \(aka Streaming Datasets\)](#) for continuous incremental execution of structured queries.
3. Non-programmers will likely use SQL as their query language through direct integration with Hive
4. JDBC/ODBC fans can use JDBC interface (through [Thrift JDBC/ODBC Server](#)) and connect their tools to Spark's distributed query engine.

Spark SQL comes with a uniform interface for data access in distributed storage systems like Cassandra or HDFS (Hive, Parquet, JSON) using specialized [DataFrameReader](#) and [DataFrameWriter](#) objects.

Spark SQL allows you to execute SQL-like queries on large volume of data that can live in Hadoop HDFS or Hadoop-compatible file systems like S3. It can access data from different data sources - files or tables.

Spark SQL defines the following types of functions:

- [standard functions](#) or [User-Defined Functions \(UDFs\)](#) that take values from a single row as input to generate a single return value for every input row.
- [basic aggregate functions](#) that operate on a group of rows and calculate a single return value per group.
- [window aggregate functions](#) that operate on a group of rows and calculate a single return value for each row in a group.

There are two supported **catalog** implementations — `in-memory` (default) and `hive` — that you can set using [spark.sql.catalogImplementation](#) property.

From user@spark:

If you already loaded csv data into a dataframe, why not register it as a table, and use Spark SQL to find max/min or any other aggregates? `SELECT MAX(column_name) FROM dftable_name ...` seems natural.

you're more comfortable with SQL, it might worth registering this DataFrame as a table and generating SQL query to it (generate a string with a series of min-max calls)

You can parse data from external data sources and let the *schema inferencer* to deduct the schema.

```
// Example 1
val df = Seq(1 -> 2).toDF("i", "j")
val query = df.groupBy('i)
  .agg(max('j).as("aggOrdering"))
  .orderBy(sum('j))
  .as[(Int, Int)]
query.collect contains (1, 2) // true

// Example 2
val df = Seq((1, 1), (-1, 1)).toDF("key", "value")
df.createOrReplaceTempView("src")
scala> sql("SELECT IF(a > 0, a, 0) FROM (SELECT key a FROM src) temp").show
+-----+
|(IF((a > 0), a, 0))|
+-----+
|          1|
|          0|
+-----+
```

Further Reading and Watching

1. [Spark SQL home page](#)
2. (video) [Spark's Role in the Big Data Ecosystem - Matei Zaharia](#)
3. [Introducing Apache Spark 2.0](#)

Datasets vs DataFrames vs RDDs

Many may have been asking yourself why they should be using Datasets rather than the foundation of all Spark - RDDs using case classes.

This document collects advantages of `Dataset` vs `RDD[CaseClass]` to answer the question [Dan has asked on twitter](#):

"In #Spark, what is the advantage of a DataSet over an RDD[CaseClass]?"

Saving to or Writing from Data Sources

With Dataset API, loading data from a data source or saving it to one is as simple as using `SparkSession.read` or `Dataset.write` methods, appropriately.

Accessing Fields / Columns

You `select` columns in a datasets without worrying about the positions of the columns.

In RDD, you have to do an additional hop over a case class and access fields by name.

Dataset API vs SQL

Spark SQL supports two "modes" to write structured queries: [Dataset API](#) and [SQL](#).

It turns out that some structured queries can be expressed easier using Dataset API, but there are some that are only possible in SQL. In other words, you may find mixing Dataset API and SQL modes challenging yet rewarding.

You could at some point consider writing structured queries using [Catalyst data structures](#) directly hoping to avoid the differences and focus on what is supported in Spark SQL, but that could quickly become unwieldy for maintenance (i.e. finding Spark SQL developers who could be comfortable with it as well as being fairly low-level and therefore possibly too dependent on a specific Spark SQL version).

This section describes the differences between Spark SQL features to develop Spark applications using Dataset API and SQL mode.

1. [RuntimeReplaceable Expressions](#) are only available using SQL mode by means of SQL functions like `nvl`, `nvl2`, `ifnull`, `nullif`, etc.
2. [Column.isin](#) and [SQL IN predicate with a subquery](#) (and [In Predicate Expression](#))

VectorizedParquetRecordReader

`VectorizedParquetRecordReader` is a concrete `SpecificParquetRecordReaderBase` for `parquet` file format for [Vectorized Parquet Decoding](#).

`VectorizedParquetRecordReader` is created exclusively when `ParquetFileFormat` is requested for a `data reader` (with `spark.sql.parquet.enableVectorizedReader` property enabled and the read schema with `AtomicType` data types only).

Note	<code>spark.sql.parquet.enableVectorizedReader</code> configuration property is enabled (<code>true</code>) by default.
------	---

`VectorizedParquetRecordReader` takes the following to be created:

- `TimeZone` (`null` when no timezone conversion is expected)
- `useOffHeap` flag (per `spark.sql.columnVector.offheap.enabled` property)
- Capacity (per `spark.sql.parquet.columnarReaderBatchSize` property)

`VectorizedParquetRecordReader` uses the `capacity` attribute for the following:

- Creating `WritableColumnVectors` when initializing a columnar batch
- Controlling number of rows when `nextBatch`

`VectorizedParquetRecordReader` uses `OFF_HEAP` memory mode when `spark.sql.columnVector.offheap.enabled` internal configuration property is enabled (`true`).

Note	<code>spark.sql.columnVector.offheap.enabled</code> configuration property is disabled (<code>false</code>) by default.
------	---

Table 1. VectorizedParquetRecordReader's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>batchIdx</code>	Current batch index that is the index of an <code>InternalRow</code> in the <code>ColumnarBatch</code> . Used when <code>VectorizedParquetRecordReader</code> is requested to <code>getCurrentValue</code> with the <code>returnColumnarBatch</code> flag disabled Starts at <code>0</code> Increments every <code>nextKeyValue</code> Reset to <code>0</code> when reading next rows into a columnar batch
<code>columnarBatch</code>	<code>ColumnarBatch</code>

columnReaders	VectorizedColumnReaders (one reader per column) to read rows as batches Initialized when <code>checkEndOfRowGroup</code> (when requested to read next rows into a columnar batch)
columnVectors	Allocated <code>WritableColumnVectors</code>
MEMORY_MODE	Memory mode of the <code>ColumnarBatch</code> <ul style="list-style-type: none"> • <code>OFF_HEAP</code> (when <code>useOffHeap</code> is on as per <code>spark.sql.columnVector.offheap.enabled</code> configuration property) • <code>ON_HEAP</code> Used exclusively when <code>VectorizedParquetRecordReader</code> is requested to <code>initBatch</code> .
missingColumns	Bitmap of columns (per index) that are missing (or simply the ones that the reader should not read)
numBatched	
returnColumnarBatch	Optimization flag to control whether <code>VectorizedParquetRecordReader</code> offers rows as the <code>ColumnarBatch</code> or one row at a time only Default: <code>false</code> Enabled (<code>true</code>) when <code>VectorizedParquetRecordReader</code> is requested to enable returning batches Used in <code>nextKeyValue</code> (to read next rows into a columnar batch) and <code>getCurrentValue</code> (to return the internal <code>ColumnarBatch</code> not a single <code>InternalRow</code>)
rowsReturned	Number of rows read already
totalCountLoadedSoFar	
totalRowCount	Total number of rows to be read

nextKeyValue Method

```
boolean nextKeyValue() throws IOException
```

Note

`nextKeyValue` is part of Hadoop's `RecordReader` to read (key, value) pairs from a Hadoop `InputSplit` to present a record-oriented view.

`nextKeyValue ...FIXME`**Note**`nextKeyValue` is used when:

- `NewHadoopRDD` is requested to compute a partition (`compute`)
- `RecordReaderIterator` is requested to check whether or not there are more internal rows

resultBatch Method

`ColumnarBatch resultBatch()``resultBatch` gives `columnarBatch` if available or does `initBatch`.**Note**`resultBatch` is used exclusively when `VectorizedParquetRecordReader` is requested to `nextKeyValue`.

Initializing — initialize Method

`void initialize(InputSplit inputSplit, TaskAttemptContext taskAttemptContext)`**Note**`initialize` is part of `SpecificParquetRecordReaderBase` Contract to...FIXME.`initialize ...FIXME`

enableReturningBatches Method

`void enableReturningBatches()``enableReturningBatches` simply turns `returnColumnarBatch` internal flag on.**Note**`enableReturningBatches` is used exclusively when `ParquetFileFormat` is requested for a `data reader` (for `vectorized parquet decoding in whole-stage codegen`).

Initializing Columnar Batch — initBatch Method

```

void initBatch(StructType partitionColumns, InternalRow partitionValues) (1)
// private
private void initBatch() (2)
private void initBatch(
    MemoryMode memMode,
    StructType partitionColumns,
    InternalRow partitionValues)

```

1. Uses `MEMORY_MODE`

2. Uses `MEMORY_MODE` and no `partitionColumns` and no `partitionValues`

`initBatch` creates the batch schema that is `sparkSchema` and the input `partitionColumns` schema.

`initBatch` requests `OffHeapColumnVector` or `OnHeapColumnVector` to allocate column vectors per the input `memMode`, i.e. `OFF_HEAP` or `ON_HEAP` memory modes, respectively. `initBatch` records the allocated column vectors as the internal `WritableColumnVectors`.

Note

`spark.sql.columnVector.offheap.enabled` configuration property controls `OFF_HEAP` or `ON_HEAP` memory modes, i.e. `true` or `false`, respectively. `spark.sql.columnVector.offheap.enabled` is disabled by default which means that `OnHeapColumnVector` is used.

`initBatch` creates a `ColumnarBatch` (with the allocated `WritableColumnVectors`) and records it as the internal `ColumnarBatch`.

`initBatch` creates new slots in the allocated `WritableColumnVectors` for the input `partitionColumns` and sets the input `partitionValues` as constants.

`initBatch` initializes `missing columns` with `nulls`.

Note

`initBatch` is used when:

- `VectorizedParquetRecordReader` is requested for `resultBatch`
- `ParquetFileFormat` is requested to build a data reader with partition column values appended

Reading Next Rows Into Columnar Batch — `nextBatch` Method

```
boolean nextBatch() throws IOException
```

`nextBatch` reads at least `capacity` rows and returns `true` when there are rows available. Otherwise, `nextBatch` returns `false` (to "announce" there are no rows available).

Internally, `nextBatch` firstly requests every `WritableColumnVector` (in the `columnVectors` internal registry) to `reset itself`.

`nextBatch` requests the `ColumnarBatch` to specify the number of rows (in batch) as `0` (effectively resetting the batch and making it available for reuse).

When the `rowsReturned` is greater than the `totalRowCount`, `nextBatch` finishes with (`returns`) `false` (to "announce" there are no rows available).

`nextBatch` `checkEndOfRowGroup`.

`nextBatch` calculates the number of rows left to be returned as a minimum of the `capacity` and the `totalCountLoadedSoFar` reduced by the `rowsReturned`.

`nextBatch` requests every `VectorizedColumnReader` to `readBatch` (with the number of rows left to be returned and associated `WritableColumnVector`).

Note	<code>VectorizedColumnReaders</code> use their own <code>WritableColumnVectors</code> for storing values read. The numbers of <code>VectorizedColumnReaders</code> and <code>WritableColumnVector</code> are equal.
Note	The number of rows in the internal <code>ColumnarBatch</code> matches the number of rows that <code>VectorizedColumnReaders</code> decoded and stored in corresponding <code>WritableColumnVectors</code> .

In the end, `nextBatch` registers the progress as follows:

- The number of rows read is added to the `rowsReturned` counter
- Requests the internal `ColumnarBatch` to set the number of rows (in batch) to be the number of rows read
- The `numBatched` registry is exactly the number of rows read
- The `batchIdx` registry becomes `0`

`nextBatch` finishes with (`returns`) `true` (to "announce" there are rows available).

Note	<code>nextBatch</code> is used exclusively when <code>VectorizedParquetRecordReader</code> is requested to <code>nextKeyValue</code> .
------	--

checkEndOfRowGroup Internal Method

```
void checkEndOfRowGroup() throws IOException
```

```
checkEndOfRowGroup ...FIXME
```

Note

`checkEndOfRowGroup` is used exclusively when `VectorizedParquetRecordReader` is requested to [read next rows into a columnar batch](#).

Getting Current Value (as Columnar Batch or Single InternalRow) — `getCurrentValue` Method

```
Object getCurrentValue()
```

Note

`getCurrentValue` is part of the Hadoop [RecordReader](#) Contract to break the data into key/value pairs for input to a Hadoop [Mapper](#).

`getCurrentValue` returns the entire [ColumnarBatch](#) with the `returnColumnarBatch` flag enabled (`true`) or requests it for a [single row](#) instead.

Note

`getCurrentValue` is used when:

- `NewHadoopRDD` is requested to compute a partition (`compute`)
- `RecordReaderIterator` is requested for the [next internal row](#)

VectorizedColumnReader

`VectorizedColumnReader` is a vectorized column reader that [VectorizedParquetRecordReader](#) uses for [Vectorized Parquet Decoding](#).

`VectorizedColumnReader` is [created](#) exclusively when `VectorizedParquetRecordReader` is requested to [checkEndOfRowGroup](#) (when requested to [read next rows into a columnar batch](#)).

Once [created](#), `VectorizedColumnReader` is requested to [read rows as a batch](#) (when `VectorizedParquetRecordReader` is requested to [read next rows into a columnar batch](#)).

`VectorizedColumnReader` is given a [WritableColumnVector](#) to store rows [read as a batch](#).

`VectorizedColumnReader` takes the following to be created:

- Parquet `ColumnDescriptor`
- Parquet `OriginalType`
- Parquet `PageReader`
- `TimeZone` (for timezone conversion to apply to int96 timestamps. `null` for no conversion)

Reading Rows As Batch — `readBatch` Method

```
void readBatch(
    int total,
    WritableColumnVector column) throws IOException
```

`readBatch` ...FIXME

Note

`readBatch` is used exclusively when `VectorizedParquetRecordReader` is requested to [read next rows into a columnar batch](#).

SpecificParquetRecordReaderBase — Hadoop RecordReader

`SpecificParquetRecordReaderBase` is the base Hadoop `RecordReader` for **parquet** format readers that directly materialize to τ .

Note	<code>RecordReader</code> reads $\langle \text{key}, \text{value} \rangle$ pairs from an Hadoop <code>InputSplit</code> .
------	---

Note	<code>VectorizedParquetRecordReader</code> is the one and only <code>SpecificParquetRecordReaderBase</code> that directly materialize to Java objects.
------	--

Table 1. SpecificParquetRecordReaderBase's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>sparkSchema</code>	Spark <code>schema</code> Initialized when <code>SpecificParquetRecordReaderBase</code> is requested to <code>initialize</code> (from the value of <code>org.apache.spark.sql.parquet.row.requested_schema</code> configuration as set when <code>ParquetFileFormat</code> is requested to build a data reader with partition column values appended)

initialize Method

```
void initialize(InputSplit inputSplit, TaskAttemptContext taskAttemptContext)
```

Note	<code>initialize</code> is part of <code>RecordReader Contract</code> to initialize a <code>RecordReader</code> .
------	---

`initialize ...FIXME`

ColumnVector Contract — In-Memory Columnar Data

`ColumnVector` is the contract of [in-memory columnar data](#) (of a [DataType](#)).

Table 1. ColumnVector Contract (Abstract Methods Only)

Method	Description
<code>getArray</code>	<pre>ColumnarArray getArray(int rowId)</pre> <p>Used when...FIXME</p>
<code>getBinary</code>	<pre>byte[] getBinary(int rowId)</pre> <p>Used when...FIXME</p>
<code>getBoolean</code>	<pre>boolean getBoolean(int rowId)</pre> <p>Used when...FIXME</p>
<code>getByte</code>	<pre>byte getByte(int rowId)</pre> <p>Used when...FIXME</p>
<code>getChild</code>	<pre>ColumnVector getChild(int ordinal)</pre> <p>Used when...FIXME</p>
<code>getDecimal</code>	<pre>Decimal getDecimal(int rowId, int precision, int scale)</pre> <p>Used when...FIXME</p>
<code>getDouble</code>	<pre>double getDouble(int rowId)</pre> <p>Used when...FIXME</p>

getFloat	<code>float getFloat(int rowId)</code> Used when...FIXME
getInt	<code>int getInt(int rowId)</code> Used when...FIXME
getLong	<code>long getLong(int rowId)</code> Used when...FIXME
getMap	<code>ColumnarMap getMap(int ordinal)</code> Used when...FIXME
getShort	<code>short getShort(int rowId)</code> Used when...FIXME
getUTF8String	<code>UTF8String getUTF8String(int rowId)</code> Used when...FIXME
hasNull	<code>boolean hasNull()</code> Used when OffHeapColumnVector and OnHeapColumnVector are requested to <code>putNotNulls</code>
isNullAt	<code>boolean isNullAt(int rowId)</code> Used in many places
numNulls	<code>int numNulls()</code> Used for testing purposes only

Table 2. ColumnVectors (Direct Implementations and Extensions)

ColumnVector	Description
ArrowColumnVector	
OrcColumnVector	
WritableColumnVector	Writable column vectors with off-heap and on-heap memory variants

`ColumnVector` takes a [DataType](#) of the column to be created.

Note	<code>ColumnVector</code> is a Java abstract class and cannot be created directly. It is created indirectly for the concrete ColumnVectors .
Note	<p><code>ColumnVector</code> is an Evolving contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.</p> <p>In other words, using the contract is as treading on thin ice.</p>

getInterval Final Method

```
CalendarInterval getInterval(int rowId)
```

`getInterval` ...FIXME

Note	<code>getInterval</code> is used when...FIXME
------	---

getStruct Final Method

```
ColumnarRow getStruct(int rowId)
```

`getStruct` ...FIXME

Note	<code>getStruct</code> is used when...FIXME
------	---

WritableColumnVector Contract

`WritableColumnVector` is the extension of the [ColumnVector contract](#) for [writable column vectors](#) that [FIXME](#).

Table 1. WritableColumnVector Contract (Abstract Methods Only)

Method	Description
<code>getArrayLength</code>	<pre>int getArrayLength(int rowId)</pre> <p>Used when...FIXME</p>
<code>getArrayOffset</code>	<pre>int getArrayOffset(int rowId)</pre> <p>Used when...FIXME</p>
<code>getBytesAsUTF8String</code>	<pre>UTF8String getBytesAsUTF8String(int rowId, int count)</pre> <p>Used when...FIXME</p>
<code>getDictID</code>	<pre>int getDictId(int rowId)</pre> <p>Used when...FIXME</p>
<code>putArray</code>	<pre>void putArray(int rowId, int offset, int length)</pre> <p>Used when...FIXME</p>
<code>putBoolean</code>	<pre>void putBoolean(int rowId, boolean value)</pre> <p>Used when...FIXME</p>

putBooleans	<pre>void putBooleans(int rowId, int count, boolean value)</pre>	Used when...FIXME
putByte	<pre>void putByte(int rowId, byte value)</pre>	Used when...FIXME
putByteArray	<pre>int putByteArray(int rowId, byte[] value, int offset, int count)</pre>	Used when...FIXME
putBytes	<pre>void putBytes(int rowId, int count, byte value) void putBytes(int rowId, int count, byte[] src, int srcIndex)</pre>	Used when...FIXME
putDouble	<pre>void putDouble(int rowId, double value)</pre>	Used when...FIXME
putDoubles	<pre>void putDoubles(int rowId, int count, byte[] src, int srcIndex) void putDoubles(int rowId, int count, double value) void putDoubles(int rowId, int count, double[] src, int srcIndex)</pre>	

	Used when...FIXME
<code>putFloat</code>	<pre>void putFloat(int rowId, float value)</pre>
	Used when...FIXME
<code>putFloats</code>	<pre>void putFloats(int rowId, int count, byte[] src, int srcIndex) void putFloats(int rowId, int count, float value) void putFloats(int rowId, int count, float[] src, int srcIndex)</pre>
	Used when...FIXME
<code>.putInt</code>	<pre>void putInt(int rowId, int value)</pre>
	Used when...FIXME
<code>putInts</code>	<pre>void putInts(int rowId, int count, byte[] src, int srcIndex) void putInts(int rowId, int count, int value) void putInts(int rowId, int count, int[] src, int srcIndex)</pre>
	Used when...FIXME
<code>putIntsLittleEndian</code>	<pre>void putIntsLittleEndian(int rowId, int count, byte[] src, int srcIndex)</pre>

	Used when...FIXME
putLong	<pre>void putLong(int rowId, long value)</pre>
	Used when...FIXME
putLongs	<pre>void putLongs(int rowId, int count, byte[] src, int srcIndex) void putLongs(int rowId, int count, long value) void putLongs(int rowId, int count, long[] src, int srcIndex)</pre>
	Used when...FIXME
putLongsLittleEndian	<pre>void putLongsLittleEndian(int rowId, int count, byte[] src, int srcIndex)</pre>
	Used when...FIXME
putNotNull	<pre>void putNotNull(int rowId)</pre> <p>Used when <code>WritableColumnVector</code> is requested to reset and appendNotNulls</p>
putNotNulls	<pre>void putNotNulls(int rowId, int count)</pre>
	Used when...FIXME
putNull	<pre>void putNull(int rowId)</pre>
	Used when...FIXME

putNulls	<pre>void putNulls(int rowId, int count)</pre>	Used when...FIXME
putShort	<pre>void putShort(int rowId, short value)</pre>	Used when...FIXME
putShorts	<pre>void putShorts(int rowId, int count, byte[] src, int srcIndex) void putShorts(int rowId, int count, short value) void putShorts(int rowId, int count, short[] src, int srcIndex)</pre>	Used when...FIXME
reserveInternal	<pre>void reserveInternal(int capacity)</pre>	<p>Used when:</p> <ul style="list-style-type: none"> • OffHeapColumnVector and OnHeapColumnVector are created • <code>WritableColumnVector</code> is requested to reserve memory of a given required capacity
reserveNewColumn	<pre>WritableColumnVector reserveNewColumn(int capacity, DataType type)</pre>	Used when...FIXME

Table 2. WritableColumnVectors

WritableColumnVector	Description
OffHeapColumnVector	
OnHeapColumnVector	

`WritableColumnVector` takes the following to be created:

- Number of rows to hold in a vector (aka `capacity`)
- [Data type](#) of the rows stored

Note	<code>WritableColumnVector</code> is a Java abstract class and cannot be created directly. It is created indirectly for the concrete WritableColumnVectors .
------	--

reset Method

```
void reset()
```

`reset` ...FIXME

Note	<p><code>reset</code> is used when:</p> <ul style="list-style-type: none"> • <code>OrcColumnarBatchReader</code> is requested to <code>nextBatch</code> • <code>VectorizedParquetRecordReader</code> is requested to read next rows into a columnar batch • OffHeapColumnVector and OnHeapColumnVector are created • <code>WritableColumnVector</code> is requested to reserveDictionaryIds
------	---

Reserving Memory Of Required Capacity — `reserve` Method

```
void reserve(int requiredCapacity)
```

`reserve` ...FIXME

Note

`reserve` is used when:

- `OrcColumnarBatchReader` is requested to `putRepeatingValues`, `putNonNullValues`, `putValues`, and `putDecimalWritables`
- `WritableColumnVector` is requested to *append values*

reserveDictionaryIds Method

```
WritableColumnVector reserveDictionaryIds(int capacity)
```

reserveDictionaryIds ...FIXME

Note

`reserveDictionaryIds` is used when...FIXME

appendNotNulls Final Method

```
int appendNotNulls(int count)
```

appendNotNulls ...FIXME

Note

`appendNotNulls` is used for testing purposes only.

OnHeapColumnVector

`OnHeapColumnVector` is a concrete [WritableColumnVector](#) that...FIXME

`OnHeapColumnVector` is [created](#) when:

- `OnHeapColumnVector` is requested to [allocate column vectors](#) and [reserveNewColumn](#)
- `OrcColumnarBatchReader` is requested to [initBatch](#)

Allocating Column Vectors — `allocateColumns` Static Method

```
OnHeapColumnVector[] allocateColumns(int capacity, StructType schema) (1)
OnHeapColumnVector[] allocateColumns(int capacity, StructField[] fields)
```

1. Simply converts `StructType` to `StructField[]` and calls the other `allocateColumns`. `allocateColumns` creates an array of `OnHeapColumnVector` for every field (to hold `capacity` number of elements of the [data type](#) per field).

Note	<p><code>allocateColumns</code> is used when:</p> <ul style="list-style-type: none"> • <code>AggregateHashMap</code> is created • <code>InMemoryTableScanExec</code> is requested to createAndDecompressColumn • <code>VectorizedParquetRecordReader</code> is requested to initBatch (with <code>ON_HEAP</code> memory mode) • <code>OrcColumnarBatchReader</code> is requested to initBatch (with <code>ON_HEAP</code> memory mode) • <code>ColumnVectorUtils</code> is requested to convert an iterator of rows into a single <code>ColumnBatch</code> (aka <code>toBatch</code>)
------	---

Creating OnHeapColumnVector Instance

`OnHeapColumnVector` takes the following when created:

- Number of elements to hold in a vector (aka `capacity`)
- [Data type](#) of the elements stored

When created, `OnHeapColumnVector` [reservesInternal](#) (for the given `capacity`) and [reset](#).

reserveInternal Method

```
void reserveInternal(int newCapacity)
```

Note

reserveInternal is part of [WritableColumnVector Contract](#) to...FIXME.

reserveInternal ...FIXME

reserveNewColumn Method

```
OnHeapColumnVector reserveNewColumn(int capacity, DataType type)
```

Note

reserveNewColumn is part of [WritableColumnVector Contract](#) to...FIXME.

reserveNewColumn ...FIXME

OffHeapColumnVector

`OffHeapColumnVector` is a concrete [WritableColumnVector](#) that...FIXME

Allocating Column Vectors — `allocateColumns` Static Method

```
OffHeapColumnVector[] allocateColumns(int capacity, StructType schema) (1)  
OffHeapColumnVector[] allocateColumns(int capacity, StructField[] fields)
```

1. Simply converts `StructType` to `StructField[]` and calls the other `allocateColumns`

`allocateColumns` creates an array of `OffHeapColumnVector` for every field (to hold `capacity` number of elements of the [data type](#) per field).

Note	<code>allocateColumns</code> is used when...FIXME
------	---

Vectorized Parquet Decoding (Reader)

Vectorized Parquet Decoding (aka **Vectorized Parquet Reader**) allows for reading datasets in parquet format in batches, i.e. rows are decoded in batches. That aims at improving memory locality and cache utilization.

Quoting [SPARK-12854 Vectorize Parquet reader](#):

The parquet encodings are largely designed to decode faster in batches, column by column. This can speed up the decoding considerably.

Vectorized Parquet Decoding is used exclusively when `ParquetFileFormat` is requested for a [data reader](#) when `spark.sql.parquet.enableVectorizedReader` property is enabled (`true`) and the read schema uses [AtomicTypes](#) data types only.

Vectorized Parquet Decoding uses `VectorizedParquetRecordReader` for vectorized decoding.

spark.sql.parquet.enableVectorizedReader Configuration Property

`spark.sql.parquet.enableVectorizedReader` configuration property is on by default.

```
val isParquetVectorizedReaderEnabled = spark.conf.get("spark.sql.parquet.enableVectorizedReader").toBoolean
assert(isParquetVectorizedReaderEnabled, "spark.sql.parquet.enableVectorizedReader should be enabled by default")
```

Dynamic Partition Inserts

Partitioning uses **partitioning columns** to divide a dataset into smaller chunks (based on the values of certain columns) that will be written into separate directories.

With a partitioned dataset, Spark SQL can load only the parts (partitions) that are really needed (and avoid doing filtering out unnecessary data on JVM). That leads to faster load time and more efficient memory consumption which gives a better performance overall.

With a partitioned dataset, Spark SQL can also be executed over different subsets (directories) in parallel at the same time.

Partitioned table (with single partition p1)

```
spark.range(10)
  .withColumn("p1", 'id % 2')
  .write
  .mode("overwrite")
  .partitionBy("p1")
  .saveAsTable("partitioned_table")
```

Dynamic Partition Inserts is a feature of Spark SQL that allows for executing `INSERT OVERWRITE TABLE` SQL statements over partitioned [HadoopFsRelations](#) that limits what partitions are deleted to overwrite the partitioned table (and its partitions) with new data.

Dynamic partitions are the partition columns that have no values defined explicitly in the PARTITION clause of `INSERT OVERWRITE TABLE` SQL statements (in the `partitionSpec` part).

Static partitions are the partition columns that have values defined explicitly in the PARTITION clause of `INSERT OVERWRITE TABLE` SQL statements (in the `partitionSpec` part).

```
// Borrowed from https://medium.com/@anuvrat/writing-into-dynamic-partitions-using-spark-2e2b818a007a
// Note day dynamic partition
INSERT OVERWRITE TABLE stats
PARTITION(country = 'US', year = 2017, month = 3, day)
SELECT ad, SUM(impressions), SUM(clicks), log_day
FROM impression_logs
GROUP BY ad;
```

Note	<code>INSERT OVERWRITE TABLE</code> SQL statement is translated into InsertIntoTable logical operator.
------	--

Dynamic Partition Inserts is only supported in SQL mode (for [INSERT OVERWRITE TABLE](#) SQL statements).

Dynamic Partition Inserts [is not supported](#) for non-file-based data sources, i.e. [InsertableRelations](#).

With Dynamic Partition Inserts, the behaviour of `OVERWRITE` keyword is controlled by `spark.sql.sources.partitionOverwriteMode` configuration property (default: `static`). The property controls whether Spark should delete **all** the partitions that match the partition specification regardless of whether there is data to be written to or not (`static`) or delete only those partitions that will have data written into (`dynamic`).

When the `dynamic` overwrite mode is enabled Spark will only delete the partitions for which it has data to be written to. All the other partitions remain intact.

From the [Writing Into Dynamic Partitions Using Spark](#):

Spark now writes data partitioned just as Hive would—which means only the partitions that are touched by the `INSERT` query get overwritten and the others are not touched.

Bucketing

Bucketing is an optimization technique that uses **buckets** (and **bucketing columns**) to determine data partitioning and avoid data shuffle.

The motivation is to optimize performance of a join query by avoiding shuffles (aka *exchanges*) of tables participating in the join. Bucketing results in fewer exchanges (and so stages).

Note	Bucketing can show the biggest benefit when pre-shuffled bucketed tables are used more than once as bucketing itself takes time (that you will offset executing multiple join queries later).
------	--

Bucketing is enabled by default. Spark SQL uses `spark.sql.sources.bucketing.enabled` configuration property to control whether bucketing should be enabled and used for query optimization or not.

Bucketing is used exclusively in `FileSourceScanExec` physical operator (when it is requested for the `input RDD` and to determine the `partitioning` and `ordering` of the output).

Example: SortMergeJoin of two FileScans

```

import org.apache.spark.sql.SaveMode
spark.range(10e4.toLong).write.mode(SaveMode.Overwrite).saveAsTable("t10e4")
spark.range(10e6.toLong).write.mode(SaveMode.Overwrite).saveAsTable("t10e6")

// Bucketing is enabled by default
// Let's check it out anyway
assert(spark.sessionState.conf.bucketingEnabled, "Bucketing disabled?!")

// Make sure that you don't end up with a BroadcastHashJoin and a BroadcastExchange
// For that, let's disable auto broadcasting
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)

val tables = spark.catalog.listTables.where($"name" startsWith "t10e")
scala> tables.show
+-----+-----+-----+-----+
| name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
| t10e4| default|      null|  MANAGED|     false|
| t10e6| default|      null|  MANAGED|     false|
+-----+-----+-----+-----+

val t4 = spark.table("t10e4")
val t6 = spark.table("t10e6")

assert(t4.count == 10e4)
assert(t6.count == 10e6)

// trigger execution of the join query
t4.join(t6, "id").foreach(_ => ())

```

The above join query is a fine example of a [SortMergeJoinExec](#) (aka [SortMergeJoin](#)) of two [FileSourceScanExecs](#) (aka [Scan](#)). The join query uses [ShuffleExchangeExec](#) physical operators (aka [Exchange](#)) to shuffle the table datasets for the SortMergeJoin.

Details for Query 6

Submitted Time: 2018/10/02 10:39:07

Duration: 4 s

Succeeded Jobs: 7

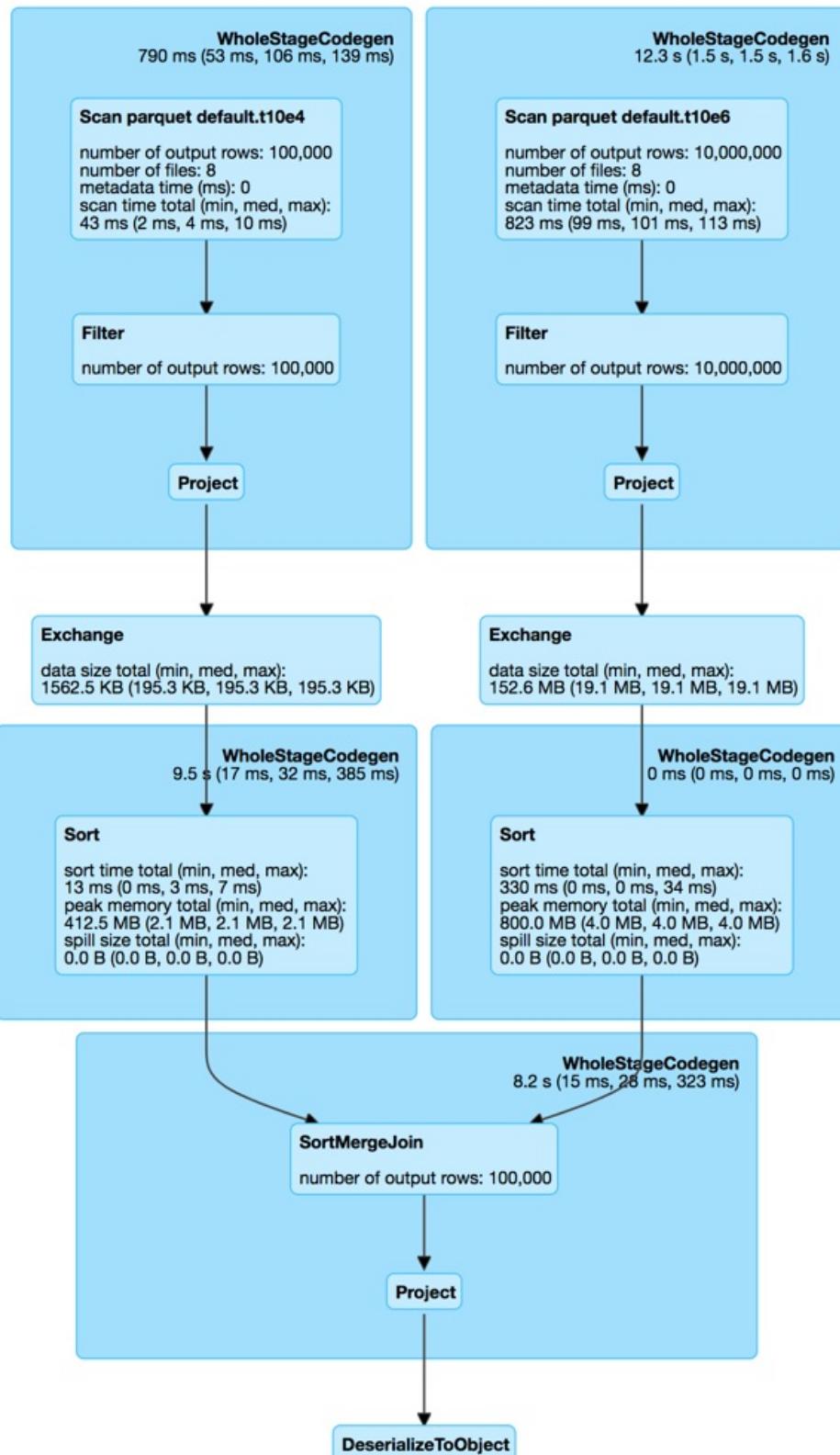


Figure 1. SortMergeJoin of FileScans (Details for Query)

One way to avoid the exchanges (and so optimize the join query) is to use table bucketing that is applicable for all file-based data sources, e.g. Parquet, ORC, JSON, CSV, that are saved as a table using [DataFrameWriter.saveAsTable](#) or simply available in a catalog by [SparkSession.table](#).

Note

Bucketing is not supported for [DataFrameWriter.save](#), [DataFrameWriter.insertInto](#) and [DataFrameWriter.jdbc](#) methods.

You use [DataFrameWriter.bucketBy](#) method to specify the number of buckets and the bucketing columns.

You can optionally sort the output rows in buckets using [DataFrameWriter.sortBy](#) method.

```
people.write
  .bucketBy(42, "name")
  .sortBy("age")
  .saveAsTable("people_bucketed")
```

Note

[DataFrameWriter.bucketBy](#) and [DataFrameWriter.sortBy](#) simply set respective internal properties that eventually become a [bucketing specification](#).

Unlike bucketing in Apache Hive, Spark SQL creates the bucket files per the number of buckets and partitions. In other words, the number of bucketing files is the number of buckets multiplied by the number of task writers (one per partition).

```
val large = spark.range(10e6.toLong)
import org.apache.spark.sql.SaveMode
large.write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode(SaveMode.Overwrite)
  .saveAsTable("bucketed_4_id")

scala> println(large.queryExecution.toRdd.getNumPartitions)
8

// That gives 8 (partitions/task writers) x 4 (buckets) = 32 files
// With _SUCCESS extra file and the ls -l header "total 794624" that gives 34 files
$ ls -tlr spark-warehouse/bucketed_4_id | wc -l
34
```

With bucketing, the Exchanges are no longer needed (as the tables are already pre-shuffled).

```
// Create bucketed tables
import org.apache.spark.sql.SaveMode
spark.range(10e4.toLong)
  .write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode(SaveMode.Overwrite)
  .saveAsTable("bucketed_4_10e4")
spark.range(10e6.toLong)
  .write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode(SaveMode.Overwrite)
  .saveAsTable("bucketed_4_10e6")

val bucketed_4_10e4 = spark.table("bucketed_4_10e4")
val bucketed_4_10e6 = spark.table("bucketed_4_10e6")

// trigger execution of the join query
bucketed_4_10e4.join(bucketed_4_10e6, "id").foreach(_ => ())
```

The above join query of the bucketed tables shows no [ShuffleExchangeExec](#) physical operators (aka *Exchange*) as the shuffling has already been executed (before the query was run).

Details for Query 11

Submitted Time: 2018/10/02 10:51:14

Duration: 1 s

Succeeded Jobs: 12

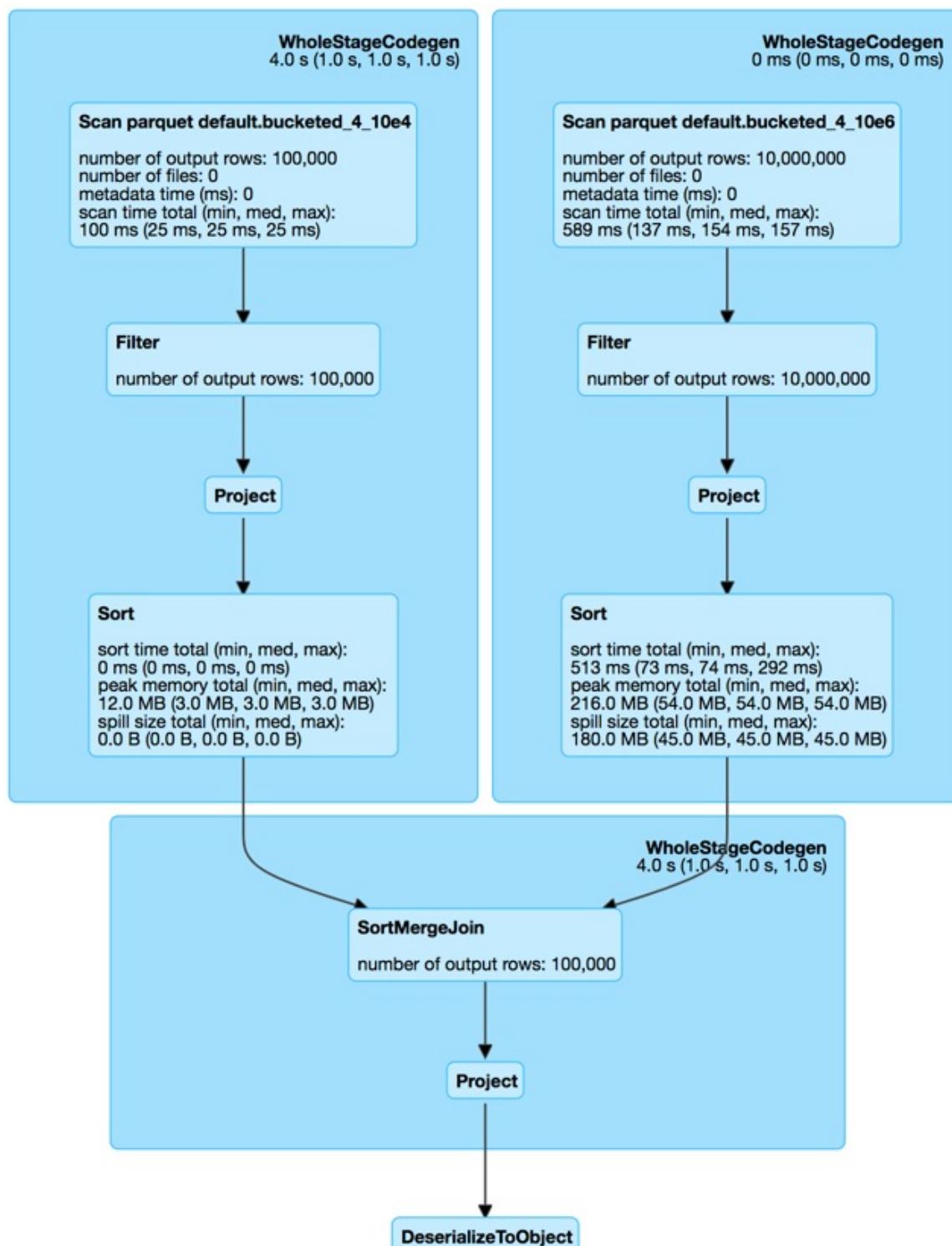


Figure 2. SortMergeJoin of Bucketed Tables (Details for Query)

The number of partitions of a bucketed table is exactly the number of buckets.

```
val bucketed_4_10e4 = spark.table("bucketed_4_10e4")
val numPartitions = bucketed_4_10e4.queryExecution.toRdd.getNumPartitions
assert(numPartitions == 4)
```

Use [SessionCatalog](#) or `DESCRIBE EXTENDED` SQL command to find the bucketing information.

```
val bucketed_tables = spark.catalog.listTables.where($"name" startsWith "bucketed_")
scala> bucketed_tables.show
+-----+-----+-----+-----+
|       name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|bucketed_4_10e4| default|      null|  MANAGED|    false|
|bucketed_4_10e6| default|      null|  MANAGED|    false|
+-----+-----+-----+-----+


val demoTable = "bucketed_4_10e4"

// DESC EXTENDED or DESC FORMATTED would also work
val describeSQL = sql(s"DESCRIBE EXTENDED $demoTable")
scala> describeSQL.show(numRows = 21, truncate = false)
+-----+
-----+-----+
|col_name           |data_type
|comment|
+-----+-----+
-----+-----+
|id                  |bigint
|      null   |
|                   |
|                   |
|# Detailed Table Information|
|                   |
|Database          |default
|                   |
|Table             |bucketed_4_10e4
|                   |
|Owner              |jacek
|                   |
|Created Time       |Tue Oct 02 10:50:50 CEST 2018
|                   |
|Last Access        |Thu Jan 01 01:00:00 CET 1970
|                   |
|Created By         |Spark 2.3.2
|                   |
|Type               |MANAGED
|                   |
|Provider            |parquet
|                   |
|Num Buckets        |4
```

```

    |   |
|Bucket Columns           |[`id`]
    |   |
|Sort Columns             |[`id`]
    |   |
|Table Properties         |[transient_lastDdlTime=1538470250]
    |   |
|Statistics                |413954 bytes
    |   |
|Location                  |file:/Users/jacek/dev/oss/spark/spark-warehouse/bucketed
_4_10e4|
|Serde Library              |org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
    |   |
|InputFormat                 |org.apache.hadoop.mapred.SequenceFileInputFormat
    |   |
|OutputFormat                |org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat
    |   |
|Storage Properties          |[serialization.format=1]
    |   |
+-----+
-----+
import org.apache.spark.sql.catalyst.TableIdentifier
val metadata = spark.sessionState.catalog.getTableMetadata(TableIdentifier(demoTable))
scala> metadata.bucketSpec.foreach(println)
4 buckets, bucket columns: [id], sort columns: [id]

```

The [number of buckets](#) has to be between `0` and `100000` exclusive or Spark SQL throws an `AnalysisException`:

```
Number of buckets should be greater than 0 but less than 100000. Got `'[numBuckets]`
```

There are however requirements that have to be met before [Spark Optimizer](#) gives a no-Exchange query plan:

1. The number of partitions on both sides of a join has to be exactly the same.
2. Both join operators have to use [HashPartitioning](#) partitioning scheme.

It is acceptable to use bucketing for one side of a join.

```

// Make sure that you don't end up with a BroadcastHashJoin and a BroadcastExchange
// For this, let's disable auto broadcasting
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)

val bucketedTableName = "bucketed_4_id"
val large = spark.range(10e5.toLong)
import org.apache.spark.sql.SaveMode
large.write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode(SaveMode.Overwrite)
  .saveAsTable(bucketedTableName)
val bucketedTable = spark.table(bucketedTableName)

val t1 = spark
  .range(4)
  .repartition(4, $"id") // Make sure that the number of partitions matches the other
  side

val q = t1.join(bucketedTable, "id")
scala> q.explain
== Physical Plan ==
*(4) Project [id#169L]
+- *(4) SortMergeJoin [id#169L], [id#167L], Inner
  :- *(2) Sort [id#169L ASC NULLS FIRST], false, 0
    :  +- Exchange hashpartitioning(id#169L, 4)
    :    +- *(1) Range (0, 4, step=1, splits=8)
  +- *(3) Sort [id#167L ASC NULLS FIRST], false, 0
    +- *(3) Project [id#167L]
      +- *(3) Filter isnotnull(id#167L)
        +- *(3) FileScan parquet default.bucketed_4_id[id#167L] Batched: true, For
mat: Parquet, Location: InMemoryFileIndex[file:/Users/jacek/dev/oss/spark/spark-wareho
use/bucketed_4_id], PartitionFilters: [], PushedFilters: [IsNotNull(id)], ReadSchema:
struct<id:bigint>

q.foreach(_ => ())

```

Details for Query 18

Submitted Time: 2018/10/02 11:02:30

Duration: 0.2 s

Succeeded Jobs: 20

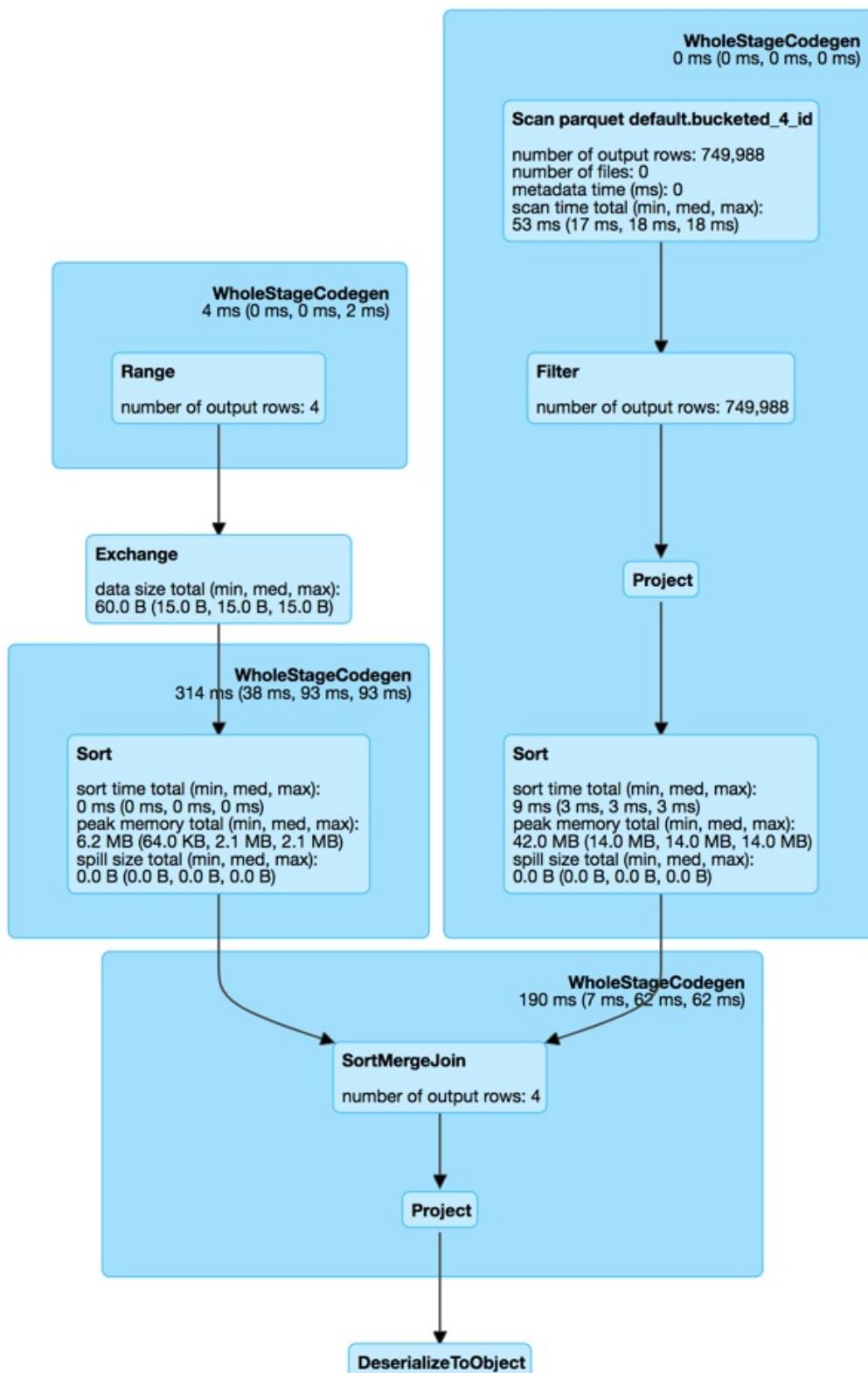


Figure 3. SortMergeJoin of One Bucketed Table (Details for Query)

Bucket Pruning — Optimizing Filtering on Bucketed Column (Reducing Bucket Files to Scan)

As of [Spark 2.4](#), Spark SQL supports **bucket pruning** to optimize filtering on bucketed column (by reducing the number of bucket files to scan).

Bucket pruning supports the following predicate expressions:

- `EqualTo` (`=`)
- `EqualNullSafe` (`<=>`)
- `In`
- `InSet`
- `And` `and` `or` of the above

[FileSourceStrategy](#) execution planning strategy is responsible for selecting only [LogicalRelations](#) over [HadoopFsRelation](#) with the [bucketing specification](#) with the following:

1. There is exactly one bucketing column
2. The number of buckets is greater than 1

Example: Bucket Pruning

```
// Enable INFO logging level of FileSourceStrategy logger to see the details of the strategy
import org.apache.spark.sql.execution.datasources.FileSourceStrategy
val logger = FileSourceStrategy.getClass.getName.replace("$", "")
import org.apache.log4j.{Level, Logger}
Logger.getLogger(logger).setLevel(Level.INFO)

val q57 = q.where($"id" isin (50, 70))
scala> val sparkPlan57 = q57.queryExecution.executedPlan
18/11/17 23:18:04 INFO FileSourceStrategy: Pruning directories with:
18/11/17 23:18:04 INFO FileSourceStrategy: Pruned 2 out of 4 buckets.
18/11/17 23:18:04 INFO FileSourceStrategy: Post-Scan Filters: id#0L IN (50,70)
18/11/17 23:18:04 INFO FileSourceStrategy: Output Data Schema: struct<id: bigint>
18/11/17 23:18:04 INFO FileSourceScanExec: Pushed Filters: In(id, [50,70])
...
scala> println(sparkPlan57.numberedTreeString)
00 *(1) Filter id#0L IN (50,70)
01 +- *(1) FileScan parquet default.bucketed_4_id[id#0L,part#1L] Batched: true, Format
: Parquet, Location: CatalogFileIndex[file:/Users/jacek/dev/oss/spark/spark-warehouse/
bucketed_4_id], PartitionCount: 2, PartitionFilters: [], PushedFilters: [In(id, [50,70
])], ReadSchema: struct<id:bigint>, SelectedBucketsCount: 2 out of 4

import org.apache.spark.sql.execution.FileSourceScanExec
val scan57 = sparkPlan57.collectFirst { case exec: FileSourceScanExec => exec }.get

import org.apache.spark.sql.execution.datasources.FileScanRDD
val rdd57 = scan57.inputRDDs.head.asInstanceOf[FileScanRDD]

import org.apache.spark.sql.execution.datasources.FilePartition
val bucketFiles57 = for {
    FilePartition(bucketId, files) <- rdd57.filePartitions
    f <- files
} yield s"Bucket $bucketId => $f"

scala> println(bucketFiles57.size)
24
```

Sorting

```

// Make sure that you don't end up with a BroadcastHashJoin and a BroadcastExchange
// Disable auto broadcasting
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)

val bucketedTableName = "bucketed_4_id"
val large = spark.range(10e5.toLong)
import org.apache.spark.sql.SaveMode
large.write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode(SaveMode.Overwrite)
  .saveAsTable(bucketedTableName)

// Describe the table and include bucketing spec only
val descSQL = sql(s"DESC FORMATTED $bucketedTableName")
  .filter($"col_name".contains("Bucket") || $"col_name" === "Sort Columns")
scala> descSQL.show
+-----+-----+-----+
|      col_name|data_type|comment|
+-----+-----+-----+
|  Num Buckets|      4|          |
|Bucket Columns| [`id`]|          |
| Sort Columns| [`id`]|          |
+-----+-----+-----+

val bucketedTable = spark.table(bucketedTableName)

val t1 = spark.range(4)
  .repartition(2, $"id") // Use just 2 partitions
  .sortWithinPartitions("id") // sort partitions

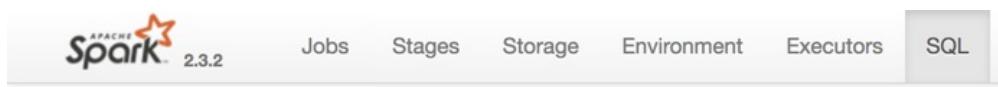
val q = t1.join(bucketedTable, "id")
// Note two exchanges and sorts
scala> q.explain
== Physical Plan ==
*(5) Project [id#205L]
+- *(5) SortMergeJoin [id#205L], [id#203L], Inner
  :- *(3) Sort [id#205L ASC NULLS FIRST], false, 0
    :  +- Exchange hashpartitioning(id#205L, 4)
    :    +- *(2) Sort [id#205L ASC NULLS FIRST], false, 0
    :      +- Exchange hashpartitioning(id#205L, 2)
    :        +- *(1) Range (0, 4, step=1, splits=8)
  +- *(4) Sort [id#203L ASC NULLS FIRST], false, 0
    +- *(4) Project [id#203L]
      +- *(4) Filter isnan(id#203L)
        +- *(4) FileScan parquet default.bucketed_4_id[id#203L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id], PartitionFilters: [], PushedFilters: [IsNotNull(id)], ReadSchema: struct<id:bigint>

q.foreach(_ => ())

```

Warning

There are two exchanges and sorts which makes the above use case almost unusable. I filed an issue at [SPARK-24025 Join of bucketed and non-bucketed tables can give two exchanges and sorts for non-bucketed side.](#)



Details for Query 23

Submitted Time: 2018/10/02 11:09:14

Duration: 0.2 s

Succeeded Jobs: 25

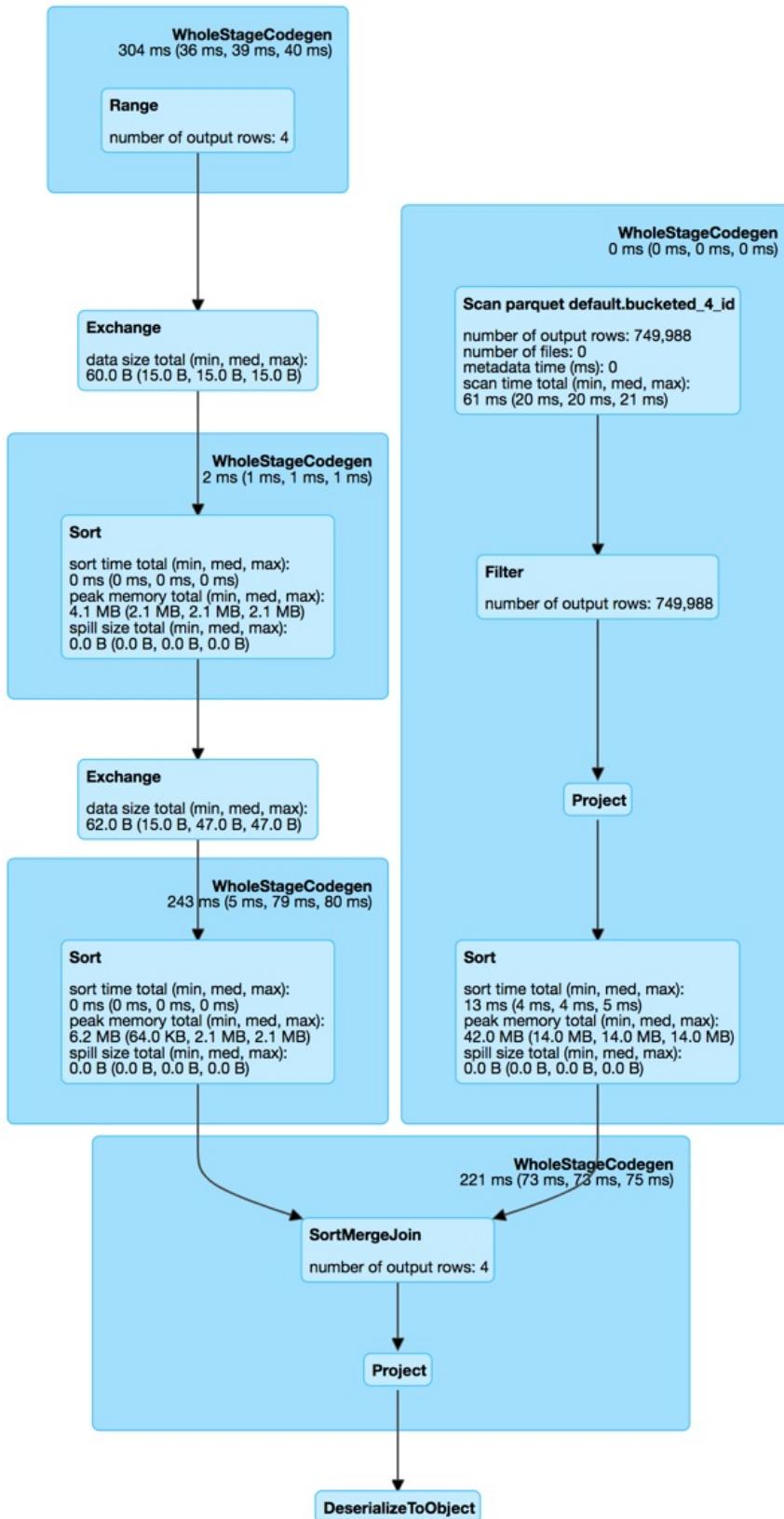


Figure 4. SortMergeJoin of Sorted Dataset and Bucketed Table (Details for Query)

spark.sql.sources.bucketing.enabled Spark SQL Configuration Property

Bucketing is enabled when `spark.sql.sources.bucketing.enabled` configuration property is turned on (`true`) and it is by default.

Tip

Use `SQLConf.bucketingEnabled` to access the current value of `spark.sql.sources.bucketing.enabled` property.

```
// Bucketing is on by default  
assert(spark.sessionState.conf.bucketingEnabled, "Bucketing disabled?!")
```

Whole-Stage Java Code Generation (Whole-Stage CodeGen)

Whole-Stage Java Code Generation (aka *Whole-Stage CodeGen*) is a physical query optimization in Spark SQL that fuses multiple physical operators (as a subtree of plans that [support code generation](#)) together into a single Java function.

Whole-Stage Java Code Generation improves the execution performance of a query by collapsing a query tree into a single optimized function that eliminates virtual function calls and leverages CPU registers for intermediate data.

	<p>Whole-Stage Code Generation is controlled by spark.sql.codegen.wholeStage Spark internal property.</p> <p>Whole-Stage Code Generation is enabled by default.</p>
Note	<pre>import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_CODEGEN_ENABLED scala> spark.conf.get(WHOLESTAGE_CODEGEN_ENABLED) res0: String = true</pre> <p>Use SQLConf.wholeStageEnabled method to access the current value.</p> <pre>scala> spark.sessionState.conf.wholeStageEnabled res1: Boolean = true</pre>

	<p>Whole-Stage Code Generation is used by some modern massively parallel processing (MPP) databases to achieve a better query execution performance.</p> <p>See Efficiently Compiling Efficient Query Plans for Modern Hardware (PDF).</p>
Note	<p>Janino is used to compile a Java source code into a Java class at runtime.</p>

Before a query is executed, [CollapseCodegenStages](#) physical preparation rule finds the physical query plans that support codegen and collapses them together as

`wholeStageCodegen` (possibly with [InputAdapter](#) in-between for physical operators with no support for Java code generation).

	<p><code>CollapseCodegenStages</code> is part of the sequence of physical preparation rules QueryExecution.preparations that will be applied in order to the physical plan before execution.</p>
Note	<p>There are the following code generation paths (as coined in this commit):</p>

1. Non-whole-stage-codegen path
1. Whole-stage-codegen "produce" path
1. Whole-stage-codegen "consume" path

Tip

Review [SPARK-12795 Whole stage codegen](#) to learn about the work to support it.

BenchmarkWholeStageCodegen — Performance Benchmark

`BenchmarkWholeStageCodegen` class provides a benchmark to measure whole stage codegen performance.

You can execute it using the command:

```
build/sbt 'sql/testOnly *BenchmarkWholeStageCodegen'
```

Note

You need to un-ignore tests in `BenchmarkWholeStageCodegen` by replacing `ignore` with `test`.

```
$ build/sbt 'sql/testOnly *BenchmarkWholeStageCodegen'
...
Running benchmark: range/limit/sum
  Running case: range/limit/sum codegen=false
22:55:23.028 WARN org.apache.hadoop.util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
  Running case: range/limit/sum codegen=true

Java HotSpot(TM) 64-Bit Server VM 1.8.0_77-b03 on Mac OS X 10.10.5
Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz

range/limit/sum:                                Best/Avg Time(ms)      Rate(M/s)    Per Row(ns)   Rel
ative
-----
-----
range/limit/sum codegen=false                  376 / 433          1394.5        0.7
1.0X
range/limit/sum codegen=true                  332 / 388          1581.3        0.6
1.1X

[info] - range/limit/sum (10 seconds, 74 milliseconds)
```


CodegenContext

CodegenContext is...FIXME

CodegenContext takes no input parameters.

```
import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext
```

CodegenContext is created when:

- WholeStageCodegenExec physical operator is requested to generate a Java source code for the child operator (when WholeStageCodegenExec is executed)
- CodeGenerator is requested for a new CodegenContext
- GenerateUnsafeRowJoiner is requested for a UnsafeRowJoiner

CodegenContext stores expressions that don't support codegen.

Example of CodegenContext.subexpressionElimination (through CodegenContext.generateExpressions)

```
import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext

// Use Catalyst DSL
import org.apache.spark.sql.catalyst.dsl.expressions._
val expressions = "hello".expr.as("world") :: "hello".expr.as("world") :: Nil

// FIXME Use a real-life query to extract the expressions

// CodegenContext.subexpressionElimination (where the elimination all happens) is a private method
// It is used exclusively in CodegenContext.generateExpressions which is public
// and does the elimination when it is enabled

// Note the doSubexpressionElimination flag is on
// Triggers the subexpressionElimination private method
ctx.generateExpressions(expressions, doSubexpressionElimination = true)

// subexpressionElimination private method uses ctx.equivalentExpressions
val commonExprs = ctx.equivalentExpressions.getAllEquivalentExprs

assert(commonExprs.length > 0, "No common expressions found")
```

Table 1. CodegenContext's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>classFunctions</code>	<p>Mutable Scala <code>Map</code> with function names, their Java source code and a class name</p> <p>New entries are added when <code>CodegenContext</code> is requested to <code>addClass</code> and <code>addNewFunctionToClass</code></p> <p>Used when <code>CodegenContext</code> is requested to <code>declareAddedFunctions</code></p>
<code>equivalentExpressions</code>	<p><code>EquivalentExpressions</code></p> <p>Expressions are <code>added</code> and then <code>fetched as equivalent sets</code> when <code>CodegenContext</code> is requested to <code>subexpressionElimination</code> (for <code>generateExpressions</code> with <code>subexpression elimination</code> enabled)</p>
<code>currentVars</code>	The list of generated columns as input of current operator
<code>INPUT_ROW</code>	The variable name of the input row of the current operator
<code>placeHolderToComments</code>	<p>Placeholders and their comments</p> <p>Used when...FIXME</p>
<code>references</code>	<p>References that are used to generate classes in the following code generators:</p> <ul style="list-style-type: none"> • <code>GenerateMutableProjection</code> • <code>GenerateOrdering</code> • <code>GeneratePredicate</code> • <code>GenerateSafeProjection</code> • <code>GenerateUnsafeProjection</code> • <code>WholeStageCodegenExec</code> • Elements are added when: <ul style="list-style-type: none"> ◦ <code>CodegenContext</code> is requested to <code>addReferenceObj</code> ◦ <code>CodegenFallback</code> is requested to <code>doGenCode</code>
<code>subExprEliminationExprs</code>	<p><code>SubExprEliminationStates</code> by <code>Expression</code></p> <p>Used when...FIXME</p>
<code>subexprFunctions</code>	Names of the functions that...FIXME

Generating Java Source Code For Code-Generated Evaluation of Multiple Expressions (With Optional Subexpression Elimination) — `generateExpressions` Method

```
generateExpressions(  
  expressions: Seq[Expression],  
  doSubexpressionElimination: Boolean = false): Seq[ExprCode]
```

(only with `subexpression elimination` enabled) `generateExpressions` does `subexpressionElimination` of the input `expressions`.

In the end, `generateExpressions` requests every expressions to `generate the Java source code for code-generated (non-interpreted) expression evaluation`.

Note	<code>generateExpressions</code> is used when:
	<ul style="list-style-type: none"> • <code>GenerateMutableProjection</code> is requested to <code>create a MutableProjection</code> • <code>GenerateUnsafeProjection</code> is requested to <code>create an ExprCode for Catalyst expressions</code> • <code>HashAggregateExec</code> is requested to <code>generate the Java source code for whole-stage consume path with grouping keys</code>

`addReferenceObj` Method

```
addReferenceObj(objName: String, obj: Any, className: String = null): String
```

`addReferenceObj ...FIXME`

Note	<code>addReferenceObj</code> is used when...FIXME
------	---

`subexpressionEliminationForWholeStageCodegen` Method

```
subexpressionEliminationForWholeStageCodegen(expressions: Seq[Expression]): SubExprCodes
```

`subexpressionEliminationForWholeStageCodegen ...FIXME`

Note

`subexpressionEliminationForWholeStageCodegen` is used exclusively when `HashAggregateExec` is requested to generate a Java source code for whole-stage consume path (with grouping keys or not).

Adding Function to Generated Class — `addNewFunction` Method

```
addNewFunction(  
    funcName: String,  
    funcCode: String,  
    inlineToOuterClass: Boolean = false): String
```

`addNewFunction ...FIXME`

Note

`addNewFunction` is used when...FIXME

subexpressionElimination Internal Method

```
subexpressionElimination(expressions: Seq[Expression]): Unit
```

`subexpressionElimination` requests `EquivalentExpressions` to `addExprTree` for every expression (in the input `expressions`).

`subexpressionElimination` requests `EquivalentExpressions` for the equivalent sets of expressions with at least two equivalent expressions (aka *common expressions*).

For every equivalent expression set, `subexpressionElimination` does the following:

1. Takes the first expression and requests it to generate a Java source code for the expression tree
2. `addNewFunction` and adds it to `subexprFunctions`
3. Creates a `SubExprEliminationState` and adds it with every common expression in the equivalent expression set to `subExprEliminationExprs`

Note

`subexpressionElimination` is used exclusively when `CodegenContext` is requested to `generateExpressions` (with `subexpression elimination` enabled).

Adding Mutable State — `addMutableState` Method

```
addMutableState(
    javaType: String,
    variableName: String,
    initFunc: String => String = _ => "",
    forceInline: Boolean = false,
    useFreshName: Boolean = true): String
```

`addMutableState ...FIXME`

```
val input = ctx.addMutableState("scala.collection.Iterator", "input", v => s"$v = inputs[0];")
```

Note

`addMutableState` is used when...FIXME

Adding Immutable State (Unless Exists Already) — `addImmutableStateIfNotExists` Method

```
addImmutableStateIfNotExists(
    javaType: String,
    variableName: String,
    initFunc: String => String = _ => ""): Unit
```

`addImmutableStateIfNotExists ...FIXME`

```
val ctx: CodegenContext = ???
val partitionMaskTerm = "partitionMask"
ctx.addImmutableStateIfNotExists(ctx.JAVA_LONG, partitionMaskTerm)
```

Note

`addImmutableStateIfNotExists` is used when...FIXME

`freshName` Method

```
freshName(name: String): String
```

`freshName ...FIXME`

Note

`freshName` is used when...FIXME

`addNewFunctionToClass` Internal Method

```
addNewFunctionToClass(  
    funcName: String,  
    funcCode: String,  
    className: String): mutable.Map[String, mutable.Map[String, String]]
```

addNewFunctionToClass ...FIXME

Note

addNewFunctionToClass is used when...FIXME

addClass Internal Method

```
addClass(className: String, classInstance: String): Unit
```

addClass ...FIXME

Note

addClass is used when...FIXME

declareAddedFunctions Method

```
declareAddedFunctions(): String
```

declareAddedFunctions ...FIXME

Note

declareAddedFunctions is used when...FIXME

declareMutableStates Method

```
declareMutableStates(): String
```

declareMutableStates ...FIXME

Note

declareMutableStates is used when...FIXME

initMutableStates Method

```
initMutableStates(): String
```

initMutableStates ...FIXME

Note	<code>initMutableStates</code> is used when...FIXME
------	---

initPartition Method

```
initPartition(): String
```

`initPartition` ...FIXME

Note	<code>initPartition</code> is used when...FIXME
------	---

emitExtraCode Method

```
emitExtraCode(): String
```

`emitExtraCode` ...FIXME

Note	<code>emitExtraCode</code> is used when...FIXME
------	---

addPartitionInitializationStatement Method

```
addPartitionInitializationStatement(statement: String): Unit
```

`addPartitionInitializationStatement` ...FIXME

Note	<code>addPartitionInitializationStatement</code> is used when...FIXME
------	---

CodeGenerator

`CodeGenerator` is a base class for generators of JVM bytecode for expression evaluation.

Table 1. CodeGenerator's Internal Properties

Name	Description
<code>cache</code>	Guava's LoadingCache with at most 100 pairs of <code>CodeAndComment</code> and <code>GeneratedClass</code> .
<code>genericMutableRowType</code>	

Tip	Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator</code> logger to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator=DEBUG</code> Refer to Logging .

CodeGenerator Contract

```
package org.apache.spark.sql.catalyst.expressions.codegen

abstract class CodeGenerator[InType, OutType] {
    def create(in: InType): OutType
    def canonicalize(in: InType): InType
    def bind(in: InType, inputSchema: Seq[Attribute]): InType
    def generate(expressions: InType, inputSchema: Seq[Attribute]): OutType
    def generate(expressions: InType): OutType
}
```

Table 2. CodeGenerator Contract

Method	Description
<code>generate</code>	Generates an evaluator for expression(s) that may (optionally) have expression(s) bound to a schema (i.e. a collection of Attribute). Used in: <ul style="list-style-type: none">• <code>ExpressionEncoder</code> for UnsafeProjection (for serialization)

Compiling Java Source Code using Janino — `doCompile` Internal Method

Caution

FIXME

Finding or Compiling Java Source Code — `compile` Method

Caution

FIXME

`create` Method

```
create(references: Seq[Expression]): UnsafeProjection
```

Caution

FIXME

Note

`create` is used when:

- `CodeGenerator` generates an expression evaluator
- `GenerateOrdering` creates a code gen ordering for `Sortorder` expressions

Creating CodegenContext — `newCodeGenContext` Method

```
newCodeGenContext(): CodegenContext
```

`newCodeGenContext` simply creates a new [CodegenContext](#).

Note

`newCodeGenContext` is used when:

- `GenerateMutableProjection` is requested to [create a MutableProjection](#)
- `GenerateOrdering` is requested to [create a BaseOrdering](#)
- `GeneratePredicate` is requested to [create a Predicate](#)
- `GenerateSafeProjection` is requested to [create a Projection](#)
- `GenerateUnsafeProjection` is requested to [create a UnsafeProjection](#)
- `GenerateColumnAccessor` is requested to [create a ColumnarIterator](#)

GenerateColumnAccessor

GenerateColumnAccessor is a [CodeGenerator](#) for...FIXME

Creating ColumnarIterator — `create` Method

```
create(columnTypes: Seq[DataType]): ColumnarIterator
```

Note

`create` is part of [CodeGenerator Contract](#) to...FIXME.

`create` ...FIXME

GenerateOrdering

GenerateOrdering is...FIXME

Creating BaseOrdering — create Method

```
create(ordering: Seq[SortOrder]): BaseOrdering  
create(schema: StructType): BaseOrdering
```

Note

create is part of [CodeGenerator Contract](#) to...FIXME.

create ...FIXME

genComparisons Method

```
genComparisons(ctx: CodegenContext, schema: StructType): String
```

genComparisons ...FIXME

Note

genComparisons is used when...FIXME

GeneratePredicate

GeneratePredicate is...FIXME

Creating Predicate — `create` Method

```
create(predicate: Expression): Predicate
```

Note

`create` is part of [CodeGenerator Contract](#) to...FIXME.

create ...FIXME

GenerateSafeProjection

GenerateSafeProjection is...FIXME

Creating Projection — `create` Method

```
create(expressions: Seq[Expression]): Projection
```

Note

`create` is part of [CodeGenerator Contract](#) to...FIXME.

create ...FIXME

BytesToBytesMap Append-Only Hash Map

BytesToBytesMap is...FIXME

- Low space overhead,
- Good memory locality, esp. for scans.

lookup Method

```
Location lookup(Object keyBase, long keyOffset, int keyLength)
Location lookup(Object keyBase, long keyOffset, int keyLength, int hash)
```

Caution	FIXME
---------	-------

safeLookup Method

```
void safeLookup(Object keyBase, long keyOffset, int keyLength, Location loc, int hash)
```

safeLookup ...FIXME

Note	safeLookup is used when BytesToBytesMap does lookup and UnsafeHashedRelation for looking up a single value or values by key.
------	--

Vectorized Query Execution (Batch Decoding)

Vectorized Query Execution (aka **Vectorized Decoding** or **Batch Decoding**) is...FIXME

ColumnarBatch — ColumnVectors as Row-Wise Table

`ColumnarBatch` allows to work with multiple `ColumnVectors` as a row-wise table.

```
import org.apache.spark.sql.types._
val schema = new StructType()
  .add("intCol", IntegerType)
  .add("doubleCol", DoubleType)
  .add("intCol2", IntegerType)
  .add("string", BinaryType)

val capacity = 4 * 1024 // 4k
import org.apache.spark.memory.MemoryMode
import org.apache.spark.sql.execution.vectorized.OnHeapColumnVector
val columns = schema.fields.map { field =>
  new OnHeapColumnVector(capacity, field.dataType)
}

import org.apache.spark.sql.vectorized.ColumnarBatch
val batch = new ColumnarBatch(columns.toArray)

// Add a row [1, 1.1, NULL]
columns(0).putInt(0, 1)
columns(1).putDouble(0, 1.1)
columns(2).putNull(0)
columns(3).putByteArray(0, "Hello".getBytes(java.nio.charset.StandardCharsets.UTF_8))
batch.setNumRows(1)

assert(batch.getRow(0).numFields == 4)
```

`ColumnarBatch` is created when:

- `InMemoryTableScanExec` physical operator is requested to `createAndDecompressColumn`
- `VectorizedParquetRecordReader` is requested to `initBatch`
- `OrcColumnarBatchReader` is requested to `initBatch`
- `ColumnVectorUtils` is requested to `toBatch`
- `ArrowPythonRunner` is requested for a `Iterator[ColumnarBatch]` (i.e. `newReaderIterator`)
- `ArrowConverters` is requested for a `ArrowRowIterator` (i.e. `fromPayloadIterator`)

`ColumnarBatch` takes an array of `ColumnVectors` to be created. `ColumnarBatch` immediately initializes the internal `MutableColumnarRow`.

The number of columns in a `ColumnarBatch` is the number of `ColumnVectors` (this batch was created with).

Note	<p><code>ColumnarBatch</code> is an <code>Evolving</code> contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.</p> <p>In other words, using the contract is as treading on thin ice.</p>
------	--

Table 1. ColumnarBatch's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>numRows</code>	Number of rows
<code>row</code>	<code>MutableColumnarRow</code> over the <code>ColumnVectors</code>

Iterator Over InternalRows (in Batch) — `rowIterator` Method

```
Iterator<InternalRow> rowIterator()
```

`rowIterator` ...FIXME

Note	<p><code>rowIterator</code> is used when:</p> <ul style="list-style-type: none"> • <code>ArrowConverters</code> is requested to <code>fromBatchIterator</code> • <code>AggregateInPandasExec</code>, <code>WindowInPandasExec</code>, and <code>FlatMapGroupsInPandasExec</code> physical operators are requested to execute (<code>doExecute</code>) • <code>ArrowEvalPythonExec</code> physical operator is requested to <code>evaluate</code>
------	---

Specifying Number of Rows (in Batch) — `setNumRows` Method

```
void setNumRows(int numRows)
```

In essence, `setNumRows` resets the batch and makes it available for reuse.

Internally, `setNumRows` simply sets the `numRows` to the given `numRows`.

Note

- `setNumRows` is used when:
- `OrcColumnarBatchReader` is requested to `nextBatch`
 - `VectorizedParquetRecordReader` is requested to `nextBatch` (when `VectorizedParquetRecordReader` is requested to `nextKeyValue`)
 - `ColumnVectorUtils` is requested to `toBatch` (for testing only)
 - `ArrowConverters` is requested to `fromBatchIterator`
 - `InMemoryTableScanExec` physical operator is requested to `createAndDecompressColumn`
 - `ArrowPythonRunner` is requested for a `ReaderIterator` (`newReaderIterator`)

Data Source API V2

Data Source API V2 (*DataSource API V2* or *DataSource V2*) is a new API for data sources in Spark SQL with the following abstractions (*contracts*):

- [DataSourceV2](#) marker interface
- [ReadSupport](#)
- [DataSourceReader](#)
- [WriteSupport](#)
- [DataSourceWriter](#)
- [SessionConfigSupport](#)
- [DataSourceV2StringFormat](#)
- [InputPartition](#)

Note

The work on Data Source API V2 was tracked under [SPARK-15689 Data source API v2](#) that was fixed in Apache Spark 2.3.0.

Note

Data Source API V2 is already heavily used in Spark Structured Streaming.

Query Planning and Execution

Data Source API V2 relies on the [DataSourceV2Strategy](#) execution planning strategy for query planning.

Data Reading

Data Source API V2 uses [DataSourceV2Relation](#) logical operator to represent data reading (aka *data scan*).

`DataSourceV2Relation` is planned (*translated*) to a [ProjectExec](#) with a [DataSourceV2ScanExec](#) physical operator (possibly under the [FilterExec](#) operator) when [DataSourceV2Strategy](#) execution planning strategy is requested to [plan a logical plan](#).

At execution, `DataSourceV2ScanExec` physical operator creates a [DataSourceRDD](#) (or a `ContinuousReader` for Spark Structured Streaming).

`DataSourceRDD` uses [InputPartitions](#) for *partitions*, *preferred locations*, and *computing partitions*.

Data Writing

Data Source API V2 uses [WriteToDataSourceV2](#) and [AppendData](#) logical operators to represent data writing (over a [DataSourceV2Relation](#) logical operator). As of Spark SQL 2.4.0, `WriteToDataSourceV2` operator was deprecated for the more specific `AppendData` operator (compare "*data writing*" to "*data append*" which is certainly more specific).

Note

One of the differences between `WriteToDataSourceV2` and `AppendData` logical operators is that the former (`WriteToDataSourceV2`) uses [DataSourceWriter](#) directly while the latter (`AppendData`) uses [DataSourceV2Relation](#) to get the [DataSourceWriter](#) from.

[WriteToDataSourceV2](#) and [AppendData](#) (with [DataSourceV2Relation](#)) logical operators are planned as (*translated to*) a [WriteToDataSourceV2Exec](#) physical operator.

At execution, `WriteToDataSourceV2Exec` physical operator...FIXME

Filter Pushdown Performance Optimization

Data Source API V2 supports **filter pushdown** performance optimization for [DataSourceReaders](#) with [SupportsPushDownFilters](#) (that is applied when [DataSourceV2Strategy](#) execution planning strategy is requested to plan a [DataSourceV2Relation](#) logical operator).

(From [Parquet Filter Pushdown](#) in Apache Drill's documentation) Filter pushdown is a performance optimization that prunes extraneous data while reading from a data source to reduce the amount of data to scan and read for queries with [supported filter expressions](#). Pruning data reduces the I/O, CPU, and network overhead to optimize query performance.

Tip

Enable INFO logging level for the [DataSourceV2Strategy logger](#) to be told [what the pushed filters are](#).

Further Reading and Watching

1. (video) [Apache Spark Data Source V2](#) by Wenchen Fan and Gengliang Wang

Subqueries (Subquery Expressions)

As of Spark 2.0, Spark SQL supports subqueries.

A **subquery** (aka **subquery expression**) is a query that is nested inside of another query.

There are the following kinds of subqueries:

1. A subquery as a source (inside a SQL `FROM` clause)
2. A scalar subquery or a predicate subquery (as a column)

Every subquery can also be **correlated** or **uncorrelated**.

A **scalar subquery** is a structured query that returns a single row and a single column only.

Spark SQL uses [ScalarSubquery \(SubqueryExpression\)](#) expression to represent scalar subqueries (while [parsing a SQL statement](#)).

```
// FIXME: ScalarSubquery in a logical plan
```

A `ScalarSubquery` expression appears as **scalar-subquery#[exprId] [conditionString]** in a logical plan.

```
// FIXME: Name of a ScalarSubquery in a logical plan
```

It is said that scalar subqueries should be used very rarely if at all and you should join instead.

Spark Analyzer uses [ResolveSubquery](#) resolution rule to [resolve subqueries](#) and at the end [makes sure that they are valid](#).

Catalyst Optimizer uses the following optimizations for subqueries:

- [PullupCorrelatedPredicates](#) optimization to [rewrite subqueries](#) and pull up correlated predicates
- [RewriteCorrelatedScalarSubquery](#) optimization to [constructLeftJoins](#)

Spark Physical Optimizer uses [PlanSubqueries](#) physical optimization to [plan queries with scalar subqueries](#).

Caution

FIXME Describe how a physical [ScalarSubquery](#) is executed (cf. `updateResult`, `eval` and `doGenCode`).

Hint Framework

Structured queries can be optimized using **Hint Framework** that allows for [specifying query hints](#).

Query hints allow for annotating a query and give a hint to the query optimizer how to optimize logical plans. This can be very useful when the query optimizer cannot make optimal decision, e.g. with respect to join methods due to conservativeness or the lack of proper statistics.

Spark SQL supports [COALESCE](#) and [REPARTITION](#) and [BROADCAST](#) hints. All remaining unresolved hints are silently removed from a query plan at [analysis](#).

Note

Hint Framework was added in [Spark SQL 2.2](#).

Specifying Query Hints

You can specify query hints using [Dataset_hint](#) operator or [SELECT SQL statements with hints](#).

```
// Dataset API
val q = spark.range(1).hint(name = "myHint", 100, true)
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- Range (0, 1, step=1, splits=Some(8))

// SQL
val q = sql("SELECT /*+ myHint (100, true) */ 1")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- 'Project [unresolvedalias(1, None)]
02     +- OneRowRelation
```

SELECT SQL Statements With Hints

`SELECT` SQL statement supports query hints as comments in SQL query that Spark SQL translates into a [UnresolvedHint](#) unary logical operator in a logical plan.

COALESCE and REPARTITION Hints

Spark SQL 2.4 added support for **COALESCE** and **REPARTITION** hints (using [SQL comments](#)):

- `SELECT /*+ COALESCE(5) */ ...`
- `SELECT /*+ REPARTITION(3) */ ...`

Broadcast Hints

Spark SQL 2.2 supports **BROADCAST** hints using [broadcast standard function](#) or [SQL comments](#):

- `SELECT /*+ MAPJOIN(b) */ ...`
- `SELECT /*+ BROADCASTJOIN(b) */ ...`
- `SELECT /*+ BROADCAST(b) */ ...`

broadcast Standard Function

While `hint` operator allows for attaching any hint to a logical plan `broadcast` standard function attaches the broadcast hint only (that actually makes it a special case of `hint` operator).

`broadcast` standard function is used for [broadcast joins \(aka map-side joins\)](#), i.e. to hint the Spark planner to broadcast a dataset regardless of the size.

```

val small = spark.range(1)
val large = spark.range(100)

// Let's use broadcast standard function first
val q = large.join(broadcast(small), "id")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'Join UsingJoin(Inner,List(id))
01 :- Range (0, 100, step=1, splits=Some(8))
02 +- ResolvedHint (broadcast)
03   +- Range (0, 1, step=1, splits=Some(8))

// Please note that broadcast standard function uses ResolvedHint not UnresolvedHint

// Let's "replicate" standard function using hint operator
// Any of the names would work (case-insensitive)
// "BROADCAST", "BROADCASTJOIN", "MAPJOIN"
val smallHinted = small.hint("broadcast")
val plan = smallHinted.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint broadcast
01 +- Range (0, 1, step=1, splits=Some(8))

// join is "clever"
// i.e. resolves UnresolvedHint into ResolvedHint immediately
val q = large.join(smallHinted, "id")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'Join UsingJoin(Inner,List(id))
01 :- Range (0, 100, step=1, splits=Some(8))
02 +- ResolvedHint (broadcast)
03   +- Range (0, 1, step=1, splits=Some(8))

```

Spark Analyzer

There are the following logical rules that [Spark Analyzer](#) uses to analyze logical plans with the [UnresolvedHint](#) logical operator:

1. [ResolveBroadcastHints](#) resolves `UnresolvedHint` operators with `BROADCAST`, `BROADCASTJOIN`, `MAPJOIN` hints to a [ResolvedHint](#)
2. [ResolveCoalesceHints](#) resolves [UnresolvedHint](#) logical operators with `COALESCE` or `REPARTITION` hints
3. [RemoveAllHints](#) simply removes all `UnresolvedHint` operators

The order of executing the above rules matters.

```
// Let's hint the query twice
// The order of hints matters as every hint operator executes Spark analyzer
// That will resolve all but the last hint
val q = spark.range(100).
  hint("broadcast").
  hint("myHint", 100, true)
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- ResolvedHint (broadcast)
02   +- Range (0, 100, step=1, splits=Some(8))

// Let's resolve unresolved hints
import org.apache.spark.sql.catalyst.rules.RuleExecutor
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.analysis.ResolveHints
import org.apache.spark.sql.internal.SQLConf
object HintResolver extends RuleExecutor[LogicalPlan] {
  lazy val batches =
    Batch("Hints", FixedPoint(maxIterations = 100),
      new ResolveHints.ResolveBroadcastHints(SQLConf.get),
      ResolveHints.RemoveAllHints) :: Nil
}
val resolvedPlan = HintResolver.execute(plan)
scala> println(resolvedPlan.numberedTreeString)
00 ResolvedHint (broadcast)
01 +- Range (0, 100, step=1, splits=Some(8))
```

Hint Operator in Catalyst DSL

You can use `hint` operator from [Catalyst DSL](#) to create a `UnresolvedHint` logical operator, e.g. for testing or Spark SQL internals exploration.

```
// Create a logical plan to add hint to
import org.apache.spark.sql.catalyst.plans.logical.LocalRelation
val r1 = LocalRelation('a.int, 'b.timestamp, 'c.boolean)
scala> println(r1.numberedTreeString)
00 LocalRelation <empty>, [a#0, b#1, c#2]

// Attach hint to the plan
import org.apache.spark.sql.catalyst.dsl.plans.-
val plan = r1.hint(name = "myHint", 100, true)
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- LocalRelation <empty>, [a#0, b#1, c#2]
```


Adaptive Query Execution

Adaptive Query Execution (aka **Adaptive Query Optimisation** or **Adaptive Optimisation**) is an optimisation of a [query execution plan](#) that [Spark Planner](#) uses for allowing alternative execution plans at runtime that would be optimized better based on runtime statistics.

Quoting the description of a [talk](#) by the authors of Adaptive Query Execution:

At runtime, the adaptive execution mode can change shuffle join to broadcast join if it finds the size of one table is less than the broadcast threshold. It can also handle skewed input data for join and change the partition number of the next stage to better fit the data scale. In general, adaptive execution decreases the effort involved in tuning SQL query parameters and improves the execution performance by choosing a better execution plan and parallelism at runtime.

Adaptive Query Execution is disabled by default. Set [spark.sql.adaptive.enabled](#) configuration property to `true` to enable it.

Note

Adaptive query execution is not supported for streaming Datasets and is disabled at their execution.

spark.sql.adaptive.enabled Configuration Property

[spark.sql.adaptive.enabled](#) configuration property turns adaptive query execution on.

Tip

Use [adaptiveExecutionEnabled](#) method to access the current value.

EnsureRequirements

[EnsureRequirements](#) is...FIXME

Further Reading and Watching

1. (video) [An Adaptive Execution Engine For Apache Spark SQL — Carson Wang](#)
2. [An adaptive execution mode for Spark SQL](#) by Carson Wang (Intel), Yucai Yu (Intel) at Strata Data Conference in Singapore, December 7, 2017

ExchangeCoordinator

`ExchangeCoordinator` is created when `EnsureRequirements` physical query optimization is requested to add an `ExchangeCoordinator` for Adaptive Query Execution.

`ExchangeCoordinator` takes the following to be created:

- Number of `ShuffleExchangeExec` unary physical operators
- Recommended size of the input data of a post-shuffle partition (configured by `spark.sql.adaptive.shuffle.targetPostShuffleInputSize` property)
- Optional advisory minimum number of post-shuffle partitions (default: `None`) (configured by `spark.sql.adaptive.minNumPostShufflePartitions` property)

`ExchangeCoordinator` keeps track of `ShuffleExchangeExec` unary physical operators that were registered (when `ShuffleExchangeExec` unary physical operator was requested to prepare itself for execution).

`ExchangeCoordinator` uses the following **text representation** (i.e. `toString`):

```
coordinator[target post-shuffle partition size: [advisoryTargetPostShuffleInputSize]]
```

postShuffleRDD Method

```
postShuffleRDD(exchange: ShuffleExchangeExec): ShuffledRowRDD
```

`postShuffleRDD` ...FIXME

Note	<code>postShuffleRDD</code> is used exclusively when <code>ShuffleExchangeExec</code> unary physical operator is requested to execute.
------	--

doEstimationIfNecessary Internal Method

```
doEstimationIfNecessary(): Unit
```

`doEstimationIfNecessary` ...FIXME

Note	<code>doEstimationIfNecessary</code> is used exclusively when <code>ExchangeCoordinator</code> is requested for a post-shuffle RDD (<code>ShuffledRowRDD</code>).
------	---

estimatePartitionStartIndices Method

```
estimatePartitionStartIndices(  
    mapOutputStatistics: Array[MapOutputStatistics]): Array[Int]
```

estimatePartitionStartIndices ...FIXME

Note	estimatePartitionStartIndices is used exclusively when ExchangeCoordinator is requested for a doEstimationIfNecessary.
------	--

registerExchange Method

```
registerExchange(exchange: ShuffleExchangeExec): Unit
```

registerExchange simply adds the [ShuffleExchangeExec](#) unary physical operator to the exchanges internal registry.

Note	registerExchange is used exclusively when ShuffleExchangeExec unary physical operator is requested to prepare itself for execution.
------	---

Subexpression Elimination In Code-Generated Expression Evaluation (Common Expression Reuse)

Subexpression Elimination (aka **Common Expression Reuse**) is an optimisation of a logical query plan that eliminates expressions in code-generated (non-interpreted) expression evaluation.

Subexpression Elimination is enabled by default. Use the internal `spark.sql.subexpressionElimination.enabled` configuration property control whether the feature is enabled (`true`) or not (`false`).

Subexpression Elimination is used (by means of `subexpressionEliminationEnabled` flag of `sparkPlan`) when the following physical operators are requested to execute (i.e. moving away from queries to an RDD of internal rows to describe a distributed computation):

- `ProjectExec`
- `HashAggregateExec` (and for `finishAggregate`)
- `ObjectHashAggregateExec`
- `SortAggregateExec`
- `WindowExec` (and creates a `lookup table` for `WindowExpressions` and factory functions for `WindowFunctionFrame`)

Internally, subexpression elimination happens when `CodegenContext` is requested for `subexpressionElimination` (when `CodegenContext` is requested to `generateExpressions` with subexpression elimination enabled).

spark.sql.subexpressionElimination.enabled Configuration Property

`spark.sql.subexpressionElimination.enabled` internal configuration property controls whether the subexpression elimination optimization is enabled or not.

Tip

Use `subexpressionEliminationEnabled` method to access the current value.

```
scala> import spark.sessionState.conf
import spark.sessionState.conf

scala> conf.subexpressionEliminationEnabled
res1: Boolean = true
```

EquivalentExpressions

`EquivalentExpressions` is...FIXME

Table 1. `EquivalentExpressions`'s Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>equivalenceMap</code>	<p>Equivalent sets of expressions, i.e. semantically equal expressions by their <code>Expr</code> "representative"</p> <p>Used when...FIXME</p>

addExprTree Method

`addExprTree(expr: Expression): Unit`

`addExprTree` ...FIXME

Note	<code>addExprTree</code> is used when <code>CodegenContext</code> is requested to subexpressionElimination or subexpressionEliminationForWholeStageCodegen .
------	--

addExpr Method

`addExpr(expr: Expression): Boolean`

`addExpr` ...FIXME

Note	<p><code>addExpr</code> is used when:</p> <ul style="list-style-type: none"> • <code>EquivalentExpressions</code> is requested to addExprTree • <code>PhysicalAggregation</code> is requested to destructure an Aggregate logical operator
------	--

Getting Equivalent Sets Of Expressions — getAllEquivalentExprs Method

`getAllEquivalentExprs: Seq[Seq[Expression]]`

`getAllEquivalentExprs` takes the values of all the equivalent sets of expressions.

Note

`getAllEquivalentExprs` is used when `CodegenContext` is requested to `subexpressionElimination` or `subexpressionEliminationForWholeStageCodegen`.

Cost-Based Optimization (CBO) of Logical Query Plan

Cost-Based Optimization (aka **Cost-Based Query Optimization** or **CBO Optimizer**) is an optimization technique in Spark SQL that uses [table statistics](#) to determine the most efficient query execution plan of a structured query (given the logical query plan).

Cost-based optimization is disabled by default. Spark SQL uses `spark.sql.cbo.enabled` configuration property to control whether the CBO should be enabled and used for query optimization or not.

Cost-Based Optimization uses [logical optimization rules](#) (e.g. `CostBasedJoinReorder`) to optimize the logical plan of a structured query based on statistics.

You first use `ANALYZE TABLE COMPUTE STATISTICS` SQL command to compute [table statistics](#). Use `DESCRIBE EXTENDED` SQL command to inspect the statistics.

Logical operators have [statistics support](#) that is used for query planning.

There is also support for [equi-height column histograms](#).

Table Statistics

The table statistics can be computed for tables, partitions and columns and are as follows:

1. **Total size** (in bytes) of a [table](#) or [table partitions](#)
2. **Row count** of a [table](#) or [table partitions](#)
3. [Column statistics](#), i.e. `min`, `max`, `num_nulls`, `distinct_count`, `avg_col_len`, `max_col_len`, `histogram`

spark.sql.cbo.enabled Spark SQL Configuration Property

Cost-based optimization is enabled when `spark.sql.cbo.enabled` configuration property is turned on, i.e. `true`.

Note

`spark.sql.cbo.enabled` configuration property is turned off, i.e. `false`, by default.

Tip

Use `SQLConf.cboEnabled` to access the current value of `spark.sql.cbo.enabled` property.

```
// CBO is disabled by default
val sqlConf = spark.sessionState.conf
scala> println(sqlConf.cboEnabled)
false

// Create a new SparkSession with CBO enabled
// You could spark-submit -c spark.sql.cbo.enabled=true
val sparkCboEnabled = spark.newSession
import org.apache.spark.sql.internal.SQLConf.CBO_ENABLED
sparkCboEnabled.conf.set(CBO_ENABLED.key, true)
val isCboEnabled = sparkCboEnabled.conf.get(CBO_ENABLED.key)
println(s"Is CBO enabled? $isCboEnabled")
```

Note

CBO is disabled explicitly in Spark Structured Streaming.

ANALYZE TABLE COMPUTE STATISTICS SQL Command

Cost-Based Optimization uses the statistics stored in a metastore (aka *external catalog*) using [ANALYZE TABLE](#) SQL command.

```
ANALYZE TABLE tableIdentifier partitionSpec?
COMPUTE STATISTICS (NOSCAN | FOR COLUMNS identifierSeq)?
```

Depending on the variant, `ANALYZE TABLE` computes different [statistics](#), i.e. of a table, partitions or columns.

1. `ANALYZE TABLE` with neither `PARTITION` specification nor `FOR COLUMNS` clause
2. `ANALYZE TABLE` with `PARTITION` specification (but no `FOR COLUMNS` clause)
3. `ANALYZE TABLE` with `FOR COLUMNS` clause (but no `PARTITION` specification)

Tip

Use `spark.sql.statistics.histogram.enabled` configuration property to enable column (equi-height) histograms that can provide better estimation accuracy but cause an extra table scan).

`spark.sql.statistics.histogram.enabled` is off by default.

Note

`ANALYZE TABLE` with `PARTITION` specification and `FOR COLUMNS` clause is incorrect.

```
// !!! INCORRECT !!!
ANALYZE TABLE t1 PARTITION (p1, p2) COMPUTE STATISTICS FOR COLUMNS id, p1
```

In such a case, `SparkSqlAstBuilder` reports a `WARN` message to the logs and skip the partition specification.

```
WARN Partition specification is ignored when collecting column statistics: [par
```

When executed, the above `ANALYZE TABLE` variants are [translated](#) to the following logical commands (in a logical query plan), respectively:

1. [AnalyzeTableCommand](#)
2. [AnalyzePartitionCommand](#)
3. [AnalyzeColumnCommand](#)

DESCRIBE EXTENDED SQL Command

You can view the statistics of a table, partitions or a column (stored in a metastore) using [DESCRIBE EXTENDED](#) SQL command.

```
(DESC | DESCRIBE) TABLE? (EXTENDED | FORMATTED)?
tableIdentifier partitionSpec? describeColName?
```

Table-level statistics are in **Statistics** row while partition-level statistics are in **Partition Statistics** row.

Tip

Use `DESC EXTENDED tableName` for table-level statistics and `DESC EXTENDED tableName PARTITION (p1, p2, ...)` for partition-level statistics only.

```
// table-level statistics are in Statistics row
scala> sql("DESC EXTENDED t1").show(numRows = 30, truncate = false)
+-----+
| col_name          | data_type |
| comment           |           |
+-----+
| id                | int       | |
|      |null|     |
| p1                | int       |
|      |null|     |
+-----+
```

```

|p2           |string
|  |null      |
|# Partition Information   |
|  |          |
|# col_name      |data_type
|  |comment|
|p1           |int
|  |null      |
|p2           |string
|  |null      |
|          |          |
|# Detailed Table Information|
|  |          |
|Database     |default
|  |          |
|Table        |t1
|  |          |
|Owner         |jacek
|  |          |
|Created Time |Wed Dec 27 14:10:44 CET 2017
|  |          |
|Last Access  |Thu Jan 01 01:00:00 CET 1970
|  |          |
|Created By   |Spark 2.3.0
|  |          |
|Type          |MANAGED
|  |          |
|Provider      |parquet
|  |          |
|Table Properties|[transient_lastDdlTime=1514453141]
|  |          |
|Statistics    |714 bytes, 2 rows
|  |          |
|Location      |file:/Users/jacek/dev/oss/spark/spark-warehouse/t1
|  |          |
|Serde Library |org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSe
rDe |          |
|InputFormat    |org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputF
ormat |          |
|OutputFormat   |org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutput
Format|          |
|Storage Properties|[serialization.format=1]
|  |          |
|Partition Provider |Catalog
|  |          |
+-----+
-----+
scala> spark.table("t1").show
+---+---+---+
| id| p1| p2|
+---+---+---+

```

```

|  0|  0|zero|
|  1|  1| one|
+---+---+---+


// partition-level statistics are in Partition Statistics row
scala> sql("DESC EXTENDED t1 PARTITION (p1=0, p2='zero')").show(numRows = 30, truncate
= false)
+-----+
-----+-----+
|col_name          |data_type
|comment|
+-----+-----+
|id                |int
|          |null |
|p1                |int
|          |null |
|p2                |string
|          |null |
|# Partition Information      |
|          |      |
|# col_name          |data_type
|comment|
|p1                |int
|          |null |
|p2                |string
|          |null |
|          |      |
|          |      |
|# Detailed Partition Information|
|          |      |
|Database          |default
|          |      |
|Table             |t1
|          |      |
|Partition Values   |[[p1=0, p2=zero]
|          |      |
|Location          |file:/Users/jacek/dev/oss/spark/spark-warehouse/t1/p
|1=0/p2=zero       |      |
|Serde Library     |org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHi
|veSerDe           |      |
|InputFormat        |org.apache.hadoop.hive.ql.io.parquet.MapredParquetIn
|putFormat          |      |
|OutputFormat       |org.apache.hadoop.hive.ql.io.parquet.MapredParquetOu
|tputFormat         |      |
|Storage Properties |[[path=file:/Users/jacek/dev/oss/spark/spark-warehou
|se/t1, serialization.format=1]]|
|Partition Parameters |{{numFiles=1, transient_lastDdlTime=1514469540, total
|Size=357}}
|Partition Statistics |357 bytes, 1 rows
|          |      |
|          |      |

```

```
|# Storage Information      |
|                           |
|Location                  |file:/Users/jacek/dev/oss/spark/spark-warehouse/t1
|                           |
|Serde Library             |org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHi
veSerDe                   |
|InputFormat                |org.apache.hadoop.hive.ql.io.parquet.MapredParquetIn
putFormat                 |
|OutputFormat               |org.apache.hadoop.hive.ql.io.parquet.MapredParquetOu
tputFormat                |
|Storage Properties         |[serialization.format=1]
|                           |
+-----+
-----+
```

You can view the statistics of a single column using `DESC EXTENDED tableName columnName` that are in a Dataset with two columns, i.e. `info_name` and `info_value`.

```
scala> sql("DESC EXTENDED t1 id").show
+-----+-----+
|info_name      |info_value|
+-----+-----+
|col_name       |id          |
|data_type      |int         |
|comment        |NULL        |
|min            |0           |
|max            |1           |
|num_nulls      |0           |
|distinct_count|2           |
|avg_col_len   |4           |
|max_col_len   |4           |
|histogram      |NULL        |
+-----+-----+
```

```
scala> sql("DESC EXTENDED t1 p1").show
+-----+-----+
|info_name      |info_value|
+-----+-----+
|col_name       |p1          |
|data_type      |int         |
|comment        |NULL        |
|min            |0           |
|max            |1           |
|num_nulls      |0           |
|distinct_count|2           |
|avg_col_len   |4           |
|max_col_len   |4           |
|histogram      |NULL        |
+-----+-----+
```

```
scala> sql("DESC EXTENDED t1 p2").show
+-----+-----+
|info_name      |info_value|
+-----+-----+
|col_name       |p2          |
|data_type      |string      |
|comment        |NULL        |
|min            |NULL        |
|max            |NULL        |
|num_nulls      |0           |
|distinct_count|2           |
|avg_col_len   |4           |
|max_col_len   |4           |
|histogram      |NULL        |
+-----+-----+
```

Cost-Based Optimizations

The [Spark Optimizer](#) uses heuristics (rules) that are applied to a logical query plan for cost-based optimization.

Among the optimization rules are the following:

1. [CostBasedJoinReorder](#) logical optimization rule for join reordering with 2 or more consecutive inner or cross joins (possibly separated by `Project` operators) when `spark.sql.cbo.enabled` and `spark.sql.cbo.joinReorder.enabled` configuration properties are both enabled.

Logical Commands for Altering Table Statistics

The following are the logical commands that [alter table statistics in a metastore](#) (aka *external catalog*):

1. [AnalyzeTableCommand](#)
2. [AnalyzeColumnCommand](#)
3. `AlterTableAddPartitionCommand`
4. `AlterTableDropPartitionCommand`
5. `AlterTableSetLocationCommand`
6. `TruncateTableCommand`
7. [InsertIntoHiveTable](#)
8. [InsertIntoHadoopFsRelationCommand](#)
9. `LoadDataCommand`

EXPLAIN COST SQL Command

Caution	FIXME See LogicalPlanStats
---------	--

LogicalPlanStats — Statistics Estimates of Logical Operator

[LogicalPlanStats](#) adds statistics support to logical operators and is used for query planning (with or without cost-based optimization, e.g. [CostBasedJoinReorder](#) or [JoinSelection](#), respectively).

Equi-Height Histograms for Columns

From [SPARK-17074](#) generate equi-height histogram for column:

Equi-height histogram is effective in handling skewed data distribution.

For equi-height histogram, the heights of all bins(intervals) are the same. The default number of bins we use is 254.

Now we use a two-step method to generate an equi-height histogram: 1. use percentile_approx to get percentiles (end points of the equi-height bin intervals); 2. use a new aggregate function to get distinct counts in each of these bins.

Note that this method takes two table scans. In the future we may provide other algorithms which need only one table scan.

From [\[SPARK-17074\] \[SQL\] Generate equi-height histogram in column statistics #19479](#):

Equi-height histogram is effective in cardinality estimation, and more accurate than basic column stats (min, max, ndv, etc) especially in skew distribution.

For equi-height histogram, all buckets (intervals) have the same height (frequency).

we use a two-step method to generate an equi-height histogram:

1. use ApproximatePercentile to get percentiles $p(0), p(1/n), p(2/n) \dots p((n-1)/n), p(1)$;
2. construct range values of buckets, e.g. $[p(0), p(1/n)], [p(1/n), p(2/n)] \dots [p((n-1)/n), p(1)]$, and use ApproxCountDistinctForIntervals to count ndv in each bucket. Each bucket is of the form: (lowerBound, higherBound, ndv).

Spark SQL uses [column statistics](#) that may optionally hold the [histogram of values](#) (which is empty by default). With [spark.sql.statistics.histogram.enabled](#) configuration property turned on [ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS](#) SQL command generates column (equi-height) histograms.

Note

`spark.sql.statistics.histogram.enabled` is off by default.

```
// Computing column statistics with histogram
// ./bin/spark-shell --conf spark.sql.statistics.histogram.enabled=true
scala> spark.sessionState.conf.histogramEnabled
res1: Boolean = true

val tableName = "t1"

// Make the example reproducible
import org.apache.spark.sql.catalyst.TableIdentifier
val tid = TableIdentifier(tableName)
val sessionCatalog = spark.sessionState.catalog
sessionCatalog.dropTable(tid, ignoreIfNotExists = true, purge = true)

// CREATE TABLE t1
Seq((0, 0, "zero"), (1, 1, "one")).
  toDF("id", "p1", "p2").
  write.
  saveAsTable(tableName)

// As we drop and create immediately we may face problems with unavailable partition files
// Invalidate cache
spark.sql(s"REFRESH TABLE $tableName")

// Use ANALYZE TABLE...FOR COLUMNS to compute column statistics
// that saves them in a metastore (aka an external catalog)
val df = spark.table(tableName)
val allCols = df.columns.mkString(",")
val analyzeTableSQL = s"ANALYZE TABLE t1 COMPUTE STATISTICS FOR COLUMNS $allCols"
spark.sql(analyzeTableSQL)

// Column statistics with histogram should be in the external catalog (metastore)
```

You can inspect the column statistics using [DESCRIBE EXTENDED](#) SQL command.

```
// Inspecting column statistics with column histogram
// See the above example for how to compute the stats
val colName = "id"
val descExtSQL = s"DESC EXTENDED $tableName $colName"

// 254 bins by default --> num_of_bins in histogram row below
scala> sql(descExtSQL).show(truncate = false)
+-----+-----+
|info_name      |info_value
+-----+-----+
|col_name       |id
|data_type      |int
|comment        |NULL
|min            |0
|max            |1
|num_nulls      |0
|distinct_count|2
|avg_col_len   |4
|max_col_len   |4
|histogram      |height: 0.007874015748031496, num_of_bins: 254
|bin_0          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_1          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_2          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_3          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_4          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_5          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_6          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_7          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_8          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_9          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
+-----+-----+
only showing top 20 rows
```

CatalogStatistics — Table Statistics From External Catalog (Metastore)

`CatalogStatistics` are **table statistics** that are stored in an [external catalog](#) (aka *metastore*):

- Physical **total size** (in bytes)
- Estimated **number of rows** (aka *row count*)
- **Column statistics** (i.e. column names and their [statistics](#))

Note	<p><code>CatalogStatistics</code> is a "subset" of the statistics in Statistics (as there are no concepts of attributes and broadcast hint in metastore).</p> <p><code>CatalogStatistics</code> are often stored in a Hive metastore and are referred as Hive statistics while <code>Statistics</code> are the Spark statistics.</p>
------	--

`CatalogStatistics` can be converted to [Spark statistics](#) using `toPlanStats` method.

`CatalogStatistics` is [created](#) when:

- [AnalyzeColumnCommand](#), [AlterTableAddPartitionCommand](#) and [TruncateTableCommand](#) commands are executed (and store statistics in [ExternalCatalog](#))
- [Commandutils](#) is requested for [updating existing table statistics](#), the [current statistics](#) (if changed)
- [HiveExternalCatalog](#) is requested for [restoring Spark statistics from properties](#) (from a Hive Metastore)
- [DetermineTableStats](#) and [PruneFileSourcePartitions](#) logical optimizations are executed (i.e. applied to a logical plan)
- [HiveClientImpl](#) is requested for a table or partition [statistics from Hive's parameters](#)

```

scala> :type spark.sessionState.catalog
org.apache.spark.sql.catalyst.catalog.SessionCatalog

// Using higher-level interface to access CatalogStatistics
// Make sure that you ran ANALYZE TABLE (as described above)
val db = spark.catalog.currentDatabase
val tableName = "t1"
val metadata = spark.sharedState.externalCatalog.getTable(db, tableName)
val stats = metadata.stats

scala> :type stats
Option[org.apache.spark.sql.catalyst.catalog.CatalogStatistics]

// Using low-level internal SessionCatalog interface to access CatalogTables
val tid = spark.sessionState.sqlParser.parseTableIdentifier(tableName)
val metadata = spark.sessionState.catalog.getTempViewOrPermanentTableMetadata(tid)
val stats = metadata.stats

scala> :type stats
Option[org.apache.spark.sql.catalyst.catalog.CatalogStatistics]

```

`CatalogStatistics` has a text representation.

```

scala> :type stats
Option[org.apache.spark.sql.catalyst.catalog.CatalogStatistics]

scala> stats.map(_.simpleString).foreach(println)
714 bytes, 2 rows

```

Converting Metastore Statistics to Spark Statistics — `toPlanStats` Method

```
toPlanStats(planOutput: Seq[Attribute], cboEnabled: Boolean): Statistics
```

`toPlanStats` converts the table statistics (from an external metastore) to [Spark statistics](#).

With [cost-based optimization](#) enabled and [row count](#) statistics available, `toPlanStats` creates a [Statistics](#) with the [estimated total \(output\) size](#), [row count](#) and column statistics.

Note	Cost-based optimization is enabled when <code>spark.sql.cbo.enabled</code> configuration property is turned on, i.e. <code>true</code> , and is disabled by default.
------	--

Otherwise, when [cost-based optimization](#) is disabled, `toPlanStats` creates a [Statistics](#) with just the mandatory [sizeInBytes](#).

Caution	FIXME Why does <code>toPlanStats</code> compute <code>sizeInBytes</code> differently per CBO?
---------	---

	<p><code>toPlanStats</code> does the reverse of HiveExternalCatalog.statsToProperties.</p>
Note	<p><code>FIXME Example</code></p>

	<p><code>toPlanStats</code> is used when HiveTableRelation and LogicalRelation are requested for statistics.</p>
--	--

ColumnStat — Column Statistics

`ColumnStat` holds the [statistics](#) of a table column (as part of the [table statistics](#) in a metastore).

Table 1. Column Statistics

Name	Description
<code>distinctCount</code>	Number of distinct values
<code>min</code>	Minimum value
<code>max</code>	Maximum value
<code>nullCount</code>	Number of <code>null</code> values
<code>avgLen</code>	Average length of the values
<code>maxLen</code>	Maximum length of the values
<code>histogram</code>	Histogram of values (as <code>Histogram</code> which is empty by default)

`ColumnStat` is computed (and [created from the result row](#)) using [ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS](#) SQL command (that `SparksSqlAstBuilder` translates to [AnalyzeColumnCommand](#) logical command).

```
val cols = "id, p1, p2"
val analyzeTableSQL = s"ANALYZE TABLE t1 COMPUTE STATISTICS FOR COLUMNS $cols"
spark.sql(analyzeTableSQL)
```

`ColumnStat` may optionally hold the [histogram of values](#) which is empty by default. With `spark.sql.statistics.histogram.enabled` configuration property turned on [ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS](#) SQL command generates column (equi-height) histograms.

Note

`spark.sql.statistics.histogram.enabled` is off by default.

You can inspect the column statistics using [DESCRIBE EXTENDED](#) SQL command.

```
scala> sql("DESC EXTENDED t1 id").show
+-----+-----+
|info_name      |info_value|
+-----+-----+
|col_name       |id          |
|data_type      |int         |
|comment        |NULL        |
|min            |0           |
|max            |1           |
|num_nulls      |0           |
|distinct_count|2           |
|avg_col_len   |4           |
|max_col_len   |4           |
|histogram      |NULL        | <-- no histogram (spark.sql.statistics.histogram.enabled off)
+-----+-----+
```

ColumnStat is part of the [statistics of a table](#).

```
// Make sure that you ran ANALYZE TABLE (as described above)
val db = spark.catalog.currentDatabase
val tableName = "t1"
val metadata = spark.sharedState.externalCatalog.getTable(db, tableName)
val stats = metadata.stats.get

scala> :type stats
org.apache.spark.sql.catalyst.catalog.CatalogStatistics

val colStats = stats.colStats
scala> :type colStats
Map[String,org.apache.spark.sql.catalyst.plans.logical.ColumnStat]
```

ColumnStat is converted to properties (serialized) while persisting the table (statistics) to a metastore.

```

scala> :type colStats
Map[String,org.apache.spark.sql.catalyst.plans.logical.ColumnStat]

val colName = "p1"

val p1stats = colStats(colName)
scala> :type p1stats
org.apache.spark.sql.catalyst.plans.logical.ColumnStat

import org.apache.spark.sql.types.DoubleType
val props = p1stats.toMap(colName, dataType = DoubleType)
scala> println(props)
Map(distinctCount -> 2, min -> 0.0, version -> 1, max -> 1.4, maxLen -> 8, avgLen -> 8
, nullCount -> 0)

```

`ColumnStat` is re-created from properties (deserialized) when `HiveExternalCatalog` is requested for restoring table statistics from properties (from a Hive Metastore).

```

scala> :type props
Map[String,String]

scala> println(props)
Map(distinctCount -> 2, min -> 0.0, version -> 1, max -> 1.4, maxLen -> 8, avgLen -> 8
, nullCount -> 0)

import org.apache.spark.sql.types.StructField
val p1 = "$p1".double

import org.apache.spark.sql.catalyst.plans.logical.ColumnStat
val colStatsOpt = ColumnStat.fromMap(table = "t1", field = p1, map = props)

scala> :type colStatsOpt
Option[org.apache.spark.sql.catalyst.plans.logical.ColumnStat]

```

`ColumnStat` is also created when `JoinEstimation` is requested to `estimateInnerOuterJoin` for `Inner`, `Cross`, `LeftOuter`, `RightOuter` and `FullOuter` joins.

```

val tableName = "t1"

// Make the example reproducible
import org.apache.spark.sql.catalyst.TableIdentifier
val tid = TableIdentifier(tableName)
val sessionCatalog = spark.sessionState.catalog
sessionCatalog.dropTable(tid, ignoreIfNotExists = true, purge = true)

// CREATE TABLE t1
Seq((0, 0, "zero"), (1, 1, "one")).
  toDF("id", "p1", "p2").
  write.
  saveAsTable(tableName)

// As we drop and create immediately we may face problems with unavailable partition files
// Invalidate cache
spark.sql(s"REFRESH TABLE $tableName")

// Use ANALYZE TABLE...FOR COLUMNS to compute column statistics
// that saves them in a metastore (aka an external catalog)
val df = spark.table(tableName)
val allCols = df.columns.mkString(",")
val analyzeTableSQL = s"ANALYZE TABLE t1 COMPUTE STATISTICS FOR COLUMNS $allCols"
spark.sql(analyzeTableSQL)

// Fetch the table metadata (with column statistics) from a metastore
val metastore = spark.sharedState.externalCatalog
val db = spark.catalog.currentDatabase
val tableMeta = metastore.getTable(db, table = tableName)

// The column statistics are part of the table statistics
val colStats = tableMeta.stats.get.colStats

scala> :type colStats
Map[String,org.apache.spark.sql.catalyst.plans.logical.ColumnStat]

scala> colStats.map { case (name, cs) => s"$name: $cs" }.foreach(println)
// the output may vary
id: ColumnStat(2,Some(0),Some(1),0,4,4,None)
p1: ColumnStat(2,Some(0),Some(1),0,4,4,None)
p2: ColumnStat(2,None,None,0,4,4,None)

```

Note

`ColumnStat` does not support `minimum` and `maximum` metrics for binary (i.e. `Array[Byte]`) and string types.

Converting Value to External/Java Representation (per Catalyst Data Type)— `toExternalString` Internal Method

```
toExternalString(v: Any, colName: String, dataType: DataType): String
```

`toExternalString` ...FIXME

Note

`toExternalString` is used exclusively when `ColumnStat` is requested for [statistic properties](#).

supportsHistogram Method

```
supportsHistogram(dataType: DataType): Boolean
```

`supportsHistogram` ...FIXME

Note

`supportsHistogram` is used when...FIXME

Converting ColumnStat to Properties (ColumnStat Serialization)— toMap Method

```
toMap(colName: String, dataType: DataType): Map[String, String]
```

`toMap` converts [ColumnStat](#) to the [properties](#).

Table 2. ColumnStat.toMap's Properties

Key	Value
<code>version</code>	1
<code>distinctCount</code>	<code>distinctCount</code>
<code>nullCount</code>	<code>nullCount</code>
<code>avgLen</code>	<code>avgLen</code>
<code>maxLen</code>	<code>maxLen</code>
<code>min</code>	External/Java representation of <code>min</code>
<code>max</code>	External/Java representation of <code>max</code>
<code>histogram</code>	Serialized version of Histogram (using <code>HistogramSerializer.serialize</code>)

Note

`toMap` adds `min`, `max`, `histogram` entries only if they are available.

Note

Interestingly, `colName` and `dataType` input parameters bring no value to `toMap` itself, but merely allow for a more user-friendly error reporting when [converting](#) `min` and `max` column statistics.

Note

`toMap` is used exclusively when `HiveExternalCatalog` is requested for [converting table statistics to properties](#) (before persisting them as part of table metadata in a Hive metastore).

Re-Creating Column Statistics from Properties (ColumnStat Deserialization) — `fromMap` Method

```
fromMap(table: String, field: StructField, map: Map[String, String]): Option[ColumnStat]
```

`fromMap` creates a `ColumnStat` by fetching [properties](#) of every [column statistic](#) from the input `map`.

`fromMap` returns `None` when recovering column statistics fails for whatever reason.

```
WARN Failed to parse column statistics for column [fieldName] in table [table]
```

Note

Interestingly, `table` input parameter brings no value to `fromMap` itself, but merely allows for a more user-friendly error reporting when parsing column statistics fails.

Note

`fromMap` is used exclusively when `HiveExternalCatalog` is requested for [restoring table statistics from properties](#) (from a Hive Metastore).

Creating Column Statistics from InternalRow (Result of Computing Column Statistics) — `rowToColumnStat` Method

```
rowToColumnStat(
  row: InternalRow,
  attr: Attribute,
  rowCount: Long,
  percentiles: Option[ArrayData]): ColumnStat
```

`rowToColumnStat` [creates](#) a `ColumnStat` from the input `row` and the following positions:

0. [distinctCount](#)
1. [min](#)
2. [max](#)
3. [nullCount](#)
4. [avgLen](#)
5. [maxLen](#)

If the 6 th field is not empty, `rowToColumnStat` uses it to create [histogram](#).

Note	<code>rowToColumnStat</code> is used exclusively when <code>AnalyzeColumnCommand</code> is executed (to compute the statistics for specified columns).
------	--

statExprs Method

```
statExprs(  
    col: Attribute,  
    conf: SQLConf,  
    colPercentiles: AttributeMap[ArrayData]): CreateNamedStruct
```

`statExprs` ...FIXME

Note	<code>statExprs</code> is used when...FIXME
------	---

EstimationUtils

`EstimationUtils` is...FIXME

getOutputSize Method

```
getOutputSize(  
    attributes: Seq[Attribute],  
    outputRowCount: BigInt,  
    attrStats: AttributeMap[ColumnStat] = AttributeMap(Nil)): BigInt
```

`getOutputSize` ...FIXME

Note	<code>getOutputSize</code> is used when...FIXME
------	---

nullColumnStat Method

```
nullColumnStat(dataType: DataType, rowCount: BigInt): ColumnStat
```

`nullColumnStat` ...FIXME

Note	<code>nullColumnStat</code> is used exclusively when <code>JoinEstimation</code> is requested to estimateInnerOuterJoin for <code>LeftOuter</code> and <code>RightOuter</code> joins.
------	---

Checking Availability of Row Count Statistic

— rowCountsExist Method

```
rowCountsExist(plans: LogicalPlan*): Boolean
```

`rowCountsExist` is positive (i.e. `true`) when every logical plan (in the input `plans`) has [estimated number of rows](#) (aka *row count*) statistic computed.

Otherwise, `rowCountsExist` is negative (i.e. `false`).

Note	<code>rowCountsExist</code> uses <code>LogicalPlanStats</code> to access the estimated statistics and query hints of a logical plan.
------	--

	<p><code>rowCountsExist</code> is used when:</p> <ul style="list-style-type: none">• <code>AggregateEstimation</code> is requested to estimate statistics and query hints of a Aggregate logical operator• <code>JoinEstimation</code> is requested to estimate statistics and query hints of a Join logical operator (regardless of the join type)• <code>ProjectEstimation</code> is requested to estimate statistics and query hints of a Project logical operator
Note	

CommandUtils — Utilities for Table Statistics

`CommandUtils` is a helper class that logical commands, e.g. `InsertInto*`, `AlterTable*Command`, `LoadDataCommand`, and CBO's `Analyze*`, use to manage table statistics.

`CommandUtils` defines the following utilities:

- Calculating Total Size of Table or Its Partitions
- Calculating Total File Size Under Path
- Creating CatalogStatistics with Current Statistics
- Updating Existing Table Statistics

Enable `INFO` logging level for `org.apache.spark.sql.execution.command.CommandUtils` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.command.CommandUtils=INFO
```

Refer to [Logging](#).

Updating Existing Table Statistics — `updateTableStats` Method

```
updateTableStats(sparkSession: SparkSession, table: CatalogTable): Unit
```

`updateTableStats` updates the table statistics of the input `CatalogTable` (only if the `statistics are available` in the metastore already).

`updateTableStats` requests `SessionCatalog` to `alterTableStats` with the `current total size` (when `spark.sql.statistics.size.autoUpdate.enabled` property is turned on) or empty statistics (that effectively removes the recorded statistics completely).

Important `updateTableStats` uses `spark.sql.statistics.size.autoUpdate.enabled` property to auto-update table statistics and can be expensive (and slow down data change commands) if the total number of files of a table is very large.

Note	<code>updateTableStats</code> uses <code>sparkSession</code> to access the current <code>SessionState</code> that it then uses to access the session-scoped <code>SessionCatalog</code> .
------	---

Note	<code>updateTableStats</code> is used when <code>InsertIntoHiveTable</code> , <code>InsertIntoHadoopFsRelationCommand</code> , <code>AlterTableDropPartitionCommand</code> , <code>AlterTableSetLocationCommand</code> and <code>LoadDataCommand</code> commands are executed.
------	--

Calculating Total Size of Table (with Partitions)

— `calculateTotalSize` Method

```
calculateTotalSize(sessionState: SessionState, catalogTable: CatalogTable): BigInt
```

`calculateTotalSize` calculates total file size for the entire input `CatalogTable` (when it has no partitions defined) or all its `partitions` (through the session-scoped `SessionCatalog`).

Note	<code>calculateTotalSize</code> uses the input <code>SessionState</code> to access the <code>SessionCatalog</code> .
------	--

Note	<p><code>calculateTotalSize</code> is used when:</p> <ul style="list-style-type: none"> • <code>AnalyzeColumnCommand</code> and <code>AnalyzeTableCommand</code> commands are executed • <code>CommandUtils</code> is requested to update existing table statistics (when <code>InsertIntoHiveTable</code>, <code>InsertIntoHadoopFsRelationCommand</code>, <code>AlterTableDropPartitionCommand</code>, <code>AlterTableSetLocationCommand</code> and <code>LoadDataCommand</code> commands are executed)
------	--

Calculating Total File Size Under Path

— `calculateLocationSize` Method

```
calculateLocationSize(
    sessionState: SessionState,
    identifier: TableIdentifier,
    locationUri: Option[URI]): Long
```

`calculateLocationSize` reads `hive.exec.stagingdir` configuration property for the staging directory (with `.hive-staging` being the default).

You should see the following INFO message in the logs:

```
INFO CommandUtils: Starting to calculate the total file size under path [locationUri].
```

`calculateLocationSize` calculates the sum of the length of all the files under the input `locationUri`.

Note

`calculateLocationSize` uses Hadoop's `FileSystem.getFileStatus` and `FileStatus.getLen` to access a file and the length of the file (in bytes), respectively.

In the end, you should see the following INFO message in the logs:

```
INFO CommandUtils: It took [durationInMs] ms to calculate the total file size under pa
th [locationUri].
```

Note

`calculateLocationSize` is used when:

- `AnalyzePartitionCommand` and `AlterTableAddPartitionCommand` commands are executed
- `CommandUtils` is requested for total size of a table or its partitions

Creating CatalogStatistics with Current Statistics

— `compareAndGetNewStats` Method

```
compareAndGetNewStats(
    oldStats: Option[CatalogStatistics],
    newTotalSize: BigInt,
    newRowCount: Option[BigInt]): Option[CatalogStatistics]
```

`compareAndGetNewStats` creates a new `CatalogStatistics` with the input `newTotalSize` and `newRowCount` only when they are different from the `oldStats`.

Note

`compareAndGetNewStats` is used when `AnalyzePartitionCommand` and `AnalyzeTableCommand` are executed.

Catalyst DSL — Implicit Conversions for Catalyst Data Structures

Catalyst DSL is a collection of [Scala implicit conversions](#) for constructing Catalyst data structures, i.e. [expressions](#) and [logical plans](#), more easily.

The goal of Catalyst DSL is to make working with Spark SQL's building blocks easier (e.g. for testing or Spark SQL internals exploration).

Table 1. Catalyst DSL's Implicit Conversions

Name	Description
ExpressionConversions	Creates expressions <ul style="list-style-type: none"> • Literals • UnresolvedAttribute and UnresolvedReference • ...
ImplicitOperators	Adds operators to expressions for complex expressions
plans	Creates logical plans <ul style="list-style-type: none"> • hint • join • table • DslLogicalPlan

Catalyst DSL is part of `org.apache.spark.sql.catalyst.dsl` package object.

```
import org.apache.spark.sql.catalyst.dsl.expressions._
scala> :type $"hello"
org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
```

Some implicit conversions from the Catalyst DSL interfere with the implicits provided automatically in `spark-shell` (through `spark.implicits._`).

```
scala> 'hello.decimal
<console>:30: error: type mismatch;
   found   : Symbol
   required: ?{def decimal: ?}
Note that implicit conversions are not applicable because they are ambiguous
both method symbolToColumn in class SQLImplicits of type (s: Symbol)org.apache.spark.sql.Column
and method DslSymbol in trait ExpressionConversions of type (sym: Symbol)
are possible conversion functions from Symbol to ?{def decimal: ?}
    'hello.decimal
    ^
<console>:30: error: value decimal is not a member of Symbol
    'hello.decimal
    ^
```

Important

Use `sbt console` with Spark libraries defined (in `build.sbt`) instead.

You can also disable an implicit conversion using a trick described in [How can I disable implicit conversions?](#)

```
// HACK: Disable symbolToColumn implicit conversion
// It is imported automatically in spark-shell (and makes demos impossible)
// implicit def symbolToColumn(s: Symbol): org.apache.spark.sql.ColumnName =
trait ThatWasABadIdea
implicit def symbolToColumn(ack: ThatWasABadIdea) = ack

// HACK: Disable $ string interpolator
// It is imported automatically in spark-shell (and makes demos impossible)
implicit class StringToColumn(val sc: StringContext) {}
```

```
import org.apache.spark.sql.catalyst.dsl.expressions._
import org.apache.spark.sql.catalyst.dsl.plans._

// ExpressionConversions

import org.apache.spark.sql.catalyst.expressions.Literal
scala> val trueLit: Literal = true
trueLit: org.apache.spark.sql.catalyst.expressions.Literal = true

import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
scala> val name: UnresolvedAttribute = 'name
name: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 'name

// NOTE: This conversion may not work, e.g. in spark-shell
```

```
// There is another implicit conversion StringToColumn in SQLImplicits
// It is automatically imported in spark-shell
// See :imports
val id: UnresolvedAttribute = $"id"

import org.apache.spark.sql.catalyst.expressions.Expression
scala> val expr: Expression = sum('id)
expr: org.apache.spark.sql.catalyst.expressions.Expression = sum('id)

// implicit class DslSymbol
scala> 'hello.s
res2: String = hello

scala> 'hello.attr
res4: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 'hello

// implicit class DslString
scala> "heло".expr
res0: org.apache.spark.sql.catalyst.expressions.Expression = heло

scala> "heло".attr
res1: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 'heло

// logical plans

scala> val t1 = table("t1")
t1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'UnresolvedRelation `t1`

scala> val p = t1.select('*).serialize[String].where('id % 2 == 0)
p: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'Filter false
+- 'SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String,
StringType, fromString, input[0, java.lang.String, true], true) AS value#1]
  +- 'Project ['*]
    +- 'UnresolvedRelation `t1`

// FIXME Does not work because SimpleAnalyzer's catalog is empty
// the p plan references a t1 table
import org.apache.spark.sql.catalyst.analysis.SimpleAnalyzer
scala> p.analyze
```

ImplicitOperators Implicit Conversions

Operators for [expressions](#), i.e. `in`.

ExpressionConversions Implicit Conversions

`ExpressionConversions` implicit conversions add [ImplicitOperators](#) operators to [Catalyst expressions](#).

Type Conversions to Literal Expressions

`ExpressionConversions` adds conversions of Scala native types (e.g. `Boolean`, `Long`, `String`, `Date`, `Timestamp`) and Spark SQL types (i.e. `Decimal`) to [Literal](#) expressions.

```
// DEMO FIXME
```

Converting Symbols to UnresolvedAttribute and AttributeReference Expressions

`ExpressionConversions` adds conversions of Scala's `Symbol` to [UnresolvedAttribute](#) and [AttributeReference](#) expressions.

```
// DEMO FIXME
```

Converting \$-Prefixed String Literals to UnresolvedAttribute Expressions

`ExpressionConversions` adds conversions of `$$"col name"` to an [UnresolvedAttribute](#) expression.

```
// DEMO FIXME
```

Adding Aggregate And Non-Aggregate Functions to Expressions

```
star(names: String*): Expression
```

`ExpressionConversions` adds the aggregate and non-aggregate functions to [Catalyst expressions](#) (e.g. `sum`, `count`, `upper`, `star`, `callFunction`, `windowSpec`, `windowExpr`)

```
import org.apache.spark.sql.catalyst.dsl.expressions._
val s = star()

import org.apache.spark.sql.catalyst.analysis.UnresolvedStar
assert(s.isInstanceOf[UnresolvedStar])

val s = star("a", "b")
scala> println(s)
WrappedArray(a, b).*
```

Creating UnresolvedFunction Expressions — `function` and `distinctFunction` Methods

`ExpressionConversions` allows creating [UnresolvedFunction](#) expressions with `function` and `distinctFunction` operators.

```
function(exprs: Expression*): UnresolvedFunction
distinctFunction(exprs: Expression*): UnresolvedFunction
```

```
import org.apache.spark.sql.catalyst.dsl.expressions._

// Works with Scala Symbols only
val f = 'f.function()
scala> :type f
org.apache.spark.sql.catalyst.analysis.UnresolvedFunction

scala> f.isDistinct
res0: Boolean = false

val g = 'g.distinctFunction()
scala> g.isDistinct
res1: Boolean = true
```

Creating AttributeReference Expressions With nullability On or Off — `notNull` and `canBeNull` Methods

`ExpressionConversions` adds `canBeNull` and `notNull` operators to create a `AttributeReference` with `nullability` turned on or off, respectively.

```
notNull: AttributeReference
canBeNull: AttributeReference
```

```
// DEMO FIXME
```

Creating BoundReference — `at` Method

```
at(ordinal: Int): BoundReference
```

`ExpressionConversions` adds `at` method to `AttributeReferences` to create [BoundReference](#) expressions.

```
import org.apache.spark.sql.catalyst.dsl.expressions._
val boundRef = 'hello.string.at(4)
scala> println(boundRef)
input[4, string, true]
```

plans Implicit Conversions for Logical Plans

Creating UnresolvedHint Logical Operator — `hint` Method

`plans` adds `hint` method to create a [UnresolvedHint](#) logical operator.

```
hint(name: String, parameters: Any*): LogicalPlan
```

Creating Join Logical Operator — `join` Method

`join` creates a [Join](#) logical operator.

```
join(
  otherPlan: LogicalPlan,
  joinType: JoinType = Inner,
  condition: Option[Expression] = None): LogicalPlan
```

Creating UnresolvedRelation Logical Operator — `table` Method

`table` creates a [UnresolvedRelation](#) logical operator.

```
table(ref: String): LogicalPlan
table(db: String, ref: String): LogicalPlan
```

```
import org.apache.spark.sql.catalyst.dsl.plans._

val t1 = table("t1")
scala> println(t1.treeString)
'UnresolvedRelation `t1`
```

DslLogicalPlan Implicit Class

```
implicit class DslLogicalPlan(val logicalPlan: LogicalPlan)
```

`DslLogicalPlan` implicit class is part of `plans` implicit conversions with extension methods (of [logical operators](#)) to build entire logical plans.

```

select(exprs: Expression*): LogicalPlan
where(condition: Expression): LogicalPlan
filter[T: Encoder](func: T => Boolean): LogicalPlan
filter[T: Encoder](func: FilterFunction[T]): LogicalPlan
serialize[T: Encoder]: LogicalPlan
deserialize[T: Encoder]: LogicalPlan
limit(limitExpr: Expression): LogicalPlan
join(
  otherPlan: LogicalPlan,
  joinType: JoinType = Inner,
  condition: Option[Expression] = None): LogicalPlan
cogroup[Key: Encoder, Left: Encoder, Right: Encoder, Result: Encoder](
  otherPlan: LogicalPlan,
  func: (Key, Iterator[Left], Iterator[Right]) => TraversableOnce[Result],
  leftGroup: Seq[Attribute],
  rightGroup: Seq[Attribute],
  leftAttr: Seq[Attribute],
  rightAttr: Seq[Attribute]): LogicalPlan
orderBy(sortExprs: SortOrder*): LogicalPlan
sortBy(sortExprs: SortOrder*): LogicalPlan
groupBy(groupingExprs: Expression*)(aggregateExprs: Expression*): LogicalPlan
window(
  windowExpressions: Seq[NamedExpression],
  partitionSpec: Seq[Expression],
  orderSpec: Seq[SortOrder]): LogicalPlan
subquery(alias: Symbol): LogicalPlan
except(otherPlan: LogicalPlan): LogicalPlan
intersect(otherPlan: LogicalPlan): LogicalPlan
union(otherPlan: LogicalPlan): LogicalPlan
generate(
  generator: Generator,
  unrequiredChildIndex: Seq[Int] = Nil,
  outer: Boolean = false,
  alias: Option[String] = None,
  outputNames: Seq[String] = Nil): LogicalPlan
insertInto(tableName: String, overwrite: Boolean = false): LogicalPlan
as(alias: String): LogicalPlan
coalesce(num: Integer): LogicalPlan
repartition(num: Integer): LogicalPlan
distribute(exprs: Expression*)(n: Int): LogicalPlan
hint(name: String, parameters: Any*): LogicalPlan

```

```
// Import plans object
// That loads implicit class DslLogicalPlan
// And so every LogicalPlan is the "target" of the DslLogicalPlan methods
import org.apache.spark.sql.catalyst.dsl.plans._

val t1 = table(ref = "t1")

// HACK: Disable symbolToColumn implicit conversion
// It is imported automatically in spark-shell (and makes demos impossible)
// implicit def symbolToColumn(s: Symbol): org.apache.spark.sql.ColumnName
trait ThatWasABadIdea
implicit def symbolToColumn(ack: ThatWasABadIdea) = ack

import org.apache.spark.sql.catalyst.dsl.expressions._
val id = 'id.long
val logicalPlan = t1.select(id)
scala> println(logicalPlan.numberedTreeString)
00 'Project [id#1L]
01 +- 'UnresolvedRelation `t1`

val t2 = table("t2")
import org.apache.spark.sql.catalyst.plans.LeftSemi
val logicalPlan = t1.join(t2, joinType = LeftSemi, condition = Some(id))
scala> println(logicalPlan.numberedTreeString)
00 'Join LeftSemi, id#1: bigint
01 :- 'UnresolvedRelation `t1`
02 +- 'UnresolvedRelation `t2`
```

Analyzing Logical Plan — `analyze` Method

`analyze: LogicalPlan`

`analyze` resolves attribute references.

`analyze` method is part of [DslLogicalPlan](#) implicit class.

Internally, `analyze` uses [EliminateSubqueryAliases](#) logical optimization and [SimpleAnalyzer](#) logical analyzer.

`// DEMO FIXME`

Fundamentals of Spark SQL Application Development

Development of a Spark SQL application requires the following steps:

1. Setting up Development Environment (IntelliJ IDEA, Scala and sbt)
2. Specifying Library Dependencies
3. Creating [SparkSession](#)
4. [Loading Data](#) from Data Sources
5. Processing Data Using [Dataset API](#)
6. [Saving Data](#) to Persistent Storage
7. Deploying Spark Application to Cluster (using `spark-submit`)

SparkSession — The Entry Point to Spark SQL

`SparkSession` is the entry point to Spark SQL. It is one of the very first objects you create while developing a Spark SQL application.

As a Spark developer, you create a `SparkSession` using the `SparkSession.builder` method (that gives you access to `Builder API` that you use to configure the session).

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder
  .appName("My Spark Application") // optional and will be autogenerated if not specified
  .master("local[*]")           // only for demo and testing purposes, use spark-submit instead
  .enableHiveSupport()          // self-explanatory, isn't it?
  .config("spark.sql.warehouse.dir", "target/spark-warehouse")
  .withExtensions { extensions =>
    extensions.injectResolutionRule { session =>
      ...
    }
    extensions.injectOptimizerRule { session =>
      ...
    }
  }
  .getOrCreate
```

Once created, `SparkSession` allows for [creating a DataFrame](#) (based on an RDD or a Scala `Seq`), [creating a Dataset](#), accessing the Spark SQL services (e.g. [ExperimentalMethods](#), [ExecutionListenerManager](#), [UDFRegistration](#)), [executing a SQL query](#), [loading a table](#) and the last but not least accessing [DataFrameReader](#) interface to load a dataset of the format of your choice (to some extent).

You can [enable Apache Hive support](#) with support for an [external Hive metastore](#).

Note	<p><code>spark</code> object in <code>spark-shell</code> (the instance of <code>SparkSession</code> that is auto-created) has Hive support enabled.</p> <p>In order to disable the pre-configured Hive support in the <code>spark</code> object, use <code>spark.sql.catalogImplementation</code> internal configuration property with <code>in-memory</code> value (that uses <code>InMemoryCatalog</code> external catalog instead).</p> <pre>\$ spark-shell --conf spark.sql.catalogImplementation=in-memory</pre>
-------------	---

You can have as many `SparkSessions` as you want in a single Spark application. The common use case is to keep relational entities separate logically in [catalogs](#) per `SparkSession`.

In the end, you stop a `SparkSession` using [SparkSession.stop](#) method.

```
spark.stop
```

Table 1. `SparkSession` API (Object and Instance Methods)

Method	Description
<code>active</code>	<code>active: SparkSession</code> (New in 2.4.0)
<code>builder</code>	<code>builder(): Builder</code> Object method to create a Builder to get the current <code>SparkSession</code> or create a new one.
<code>catalog</code>	<code>catalog: Catalog</code> Access to the current metadata catalog of relational entities, e.g. databases, functions, table columns, and temporary views.
<code>clearActiveSession</code>	<code>clearActiveSession(): Unit</code> Object method
<code>clearDefaultSession</code>	<code>clearDefaultSession(): Unit</code> Object method
<code>close</code>	<code>close(): Unit</code>
<code>conf</code>	<code>conf: RuntimeConfig</code> Access to the current runtime configuration

<code>createDataFrame</code>	<code>createDataFrame(rdd: RDD[_], beanClass: Class[_]): DataFrame</code> <code>createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame</code> <code>createDataFrame[A <: Product : TypeTag](rdd: RDD[A]): DataFrame</code> <code>createDataFrame[A <: Product : TypeTag](data: Seq[A]): DataFrame</code>
<code>createDataset</code>	<code>createDataset[T : Encoder](data: RDD[T]): Dataset[T]</code> <code>createDataset[T : Encoder](data: Seq[T]): Dataset[T]</code>
<code>emptyDataFrame</code>	<code>emptyDataFrame: DataFrame</code>
<code>emptyDataset</code>	<code>emptyDataset[T: Encoder]: Dataset[T]</code>
<code>experimental</code>	<code>experimental: ExperimentalMethods</code> Access to the current ExperimentalMethods
<code>getActiveSession</code>	<code>getActiveSession: Option[SparkSession]</code> Object method
<code>getDefaultSession</code>	<code>getDefaultSession: Option[SparkSession]</code> Object method
<code>implicits</code>	<code>import spark.implicits._</code> Implicits conversions
<code>listenerManager</code>	<code>listenerManager: ExecutionListenerManager</code> Access to the current ExecutionListenerManager
<code>newSession</code>	<code>newSession(): SparkSession</code> Creates a new <code>SparkSession</code>

range	<pre>range(end: Long): Dataset[java.lang.Long] range(start: Long, end: Long): Dataset[java.lang.Long] range(start: Long, end: Long, step: Long): Dataset[java.lang.Long] range(start: Long, end: Long, step: Long, numPartitions: Int): Dataset[java.lang.Long]</pre> <p>Creates a <code>Dataset[java.lang.Long]</code></p>
read	<pre>read: DataFrameReader</pre> <p>Access to the current DataFrameReader to load data from external sources</p>
sessionState	<pre>sessionState: SessionState</pre> <p>Access to the current SessionState</p> <p>Internally, <code>sessionState</code> clones the optional parent <code>SessionState</code> (if any), when creating the <code>SparkSession</code>) or creates a new <code>SessionState</code> using BaseSessionStateBuilder as defined by spark.sql.catalogImplementation configuration property:</p> <ul style="list-style-type: none"> • in-memory (default) for org.apache.spark.sql.internal.SessionStateBuilder • hive for org.apache.spark.sql.hive.HiveSessionStateBuilder
setActiveSession	<pre>setActiveSession(session: SparkSession): Unit</pre> <p>Object method</p>
setDefaultSession	<pre>setDefaultSession(session: SparkSession): Unit</pre> <p>Object method</p>
sharedState	<pre>sharedState: SharedState</pre> <p>Access to the current SharedState</p>
sparkContext	<pre>sparkContext: SparkContext</pre> <p>Access to the underlying <code>SparkContext</code></p>

<code>sql</code>	<pre>sql(sqlText: String): DataFrame</pre> <p>"Executes" a SQL query</p>
<code>sqlContext</code>	<pre>sqlContext: SQLContext</pre> <p>Access to the underlying SQLContext</p>
<code>stop</code>	<pre>stop(): Unit</pre> <p>Stops the associated SparkContext</p>
<code>table</code>	<pre>table(tableName: String): DataFrame</pre> <p>Loads data from a table</p>
<code>time</code>	<pre>time[T](f: => T): T</pre> <p>Executes a code block and prints out (to standard output) the time to execute it</p>
<code>udf</code>	<pre>udf: UDFRegistration</pre> <p>Access to the current UDFRegistration</p>
<code>version</code>	<pre>version: String</pre> <p>Returns the version of Apache Spark</p>
Note	<p><code>baseRelationToDataFrame</code> acts as a mechanism to plug <code>BaseRelation</code> object hierarchy in into <code>LogicalPlan</code> object hierarchy that <code>SparkSession</code> uses to bridge them.</p>

Creating SparkSession Using Builder Pattern — `builder` Object Method

```
builder(): Builder
```

`builder` creates a new [Builder](#) that you use to build a fully-configured `SparkSession` using a *fluent API*.

```
import org.apache.spark.sql.SparkSession
val builder = SparkSession.builder
```

Tip

Read about [Fluent interface](#) design pattern in Wikipedia, the free encyclopedia.

Accessing Version of Spark — `version` Method

```
version: String
```

`version` returns the version of Apache Spark in use.

Internally, `version` uses `spark.SPARK_VERSION` value that is the `version` property in `spark-version-info.properties` properties file on CLASSPATH.

Creating Empty Dataset (Given Encoder) — `emptyDataset` Operator

```
emptyDataset[T: Encoder]: Dataset[T]
```

`emptyDataset` creates an empty [Dataset](#) (assuming that future records being of type `T`).

```
scala> val strings = spark.emptyDataset[String]
strings: org.apache.spark.sql.Dataset[String] = [value: string]

scala> strings.printSchema
root
 |-- value: string (nullable = true)
```

`emptyDataset` creates a [LocalRelation](#) logical query plan.

Creating Dataset from Local Collections or RDDs — `createDataset` Methods

```
createDataset[T : Encoder](data: RDD[T]): Dataset[T]
createDataset[T : Encoder](data: Seq[T]): Dataset[T]
```

`createDataset` creates a [Dataset](#) from a local Scala collection, i.e. `seq[T]`, Java's `List[T]`, or a distributed `RDD[T]`.

```
scala> val one = spark.createDataset(Seq(1))
one: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> one.show
+---+
|value|
+---+
|    1|
+---+
```

`createDataset` creates a [LocalRelation](#) (for the input `data` collection) or [LogicalRDD](#) (for the input `RDD[T]`) logical operators.

You may want to consider `implicits` object and `toDS` method instead.

Tip

```
val spark: SparkSession = ...
import spark.implicits._

scala> val one = Seq(1).toDS
one: org.apache.spark.sql.Dataset[Int] = [value: int]
```

Internally, `createDataset` first looks up the implicit [expression encoder](#) in scope to access the `AttributeReference`s (of the [schema](#)).

Note

Only unresolved [expression encoders](#) are currently supported.

The expression encoder is then used to map elements (of the input `seq[T]`) into a collection of [InternalRows](#). With the references and rows, `createDataset` returns a [Dataset](#) with a [LocalRelation](#) [logical query plan](#).

Creating Dataset With Single Long Column — `range` Operator

```
range(end: Long): Dataset[java.lang.Long]
range(start: Long, end: Long): Dataset[java.lang.Long]
range(start: Long, end: Long, step: Long): Dataset[java.lang.Long]
range(start: Long, end: Long, step: Long, numPartitions: Int): Dataset[java.lang.Long]
```

`range` family of methods create a [Dataset](#) of `Long` numbers.

```
scala> spark.range(start = 0, end = 4, step = 2, numPartitions = 5).show
+---+
| id|
+---+
| 0|
| 2|
+---+
```

Note The three first variants (that do not specify `numPartitions` explicitly) use `SparkContext.defaultParallelism` for the number of partitions `numPartitions`.

Internally, `range` creates a new `Dataset[Long]` with `Range` logical plan and `Encoders.LONG encoder`.

Creating Empty DataFrame — `emptyDataFrame` method

```
emptyDataFrame: DataFrame
```

`emptyDataFrame` creates an empty `DataFrame` (with no rows and columns).

It calls `createDataFrame` with an empty `RDD[Row]` and an empty schema `StructType(Nil)`.

Creating DataFrames from Local Collections or RDDs — `createDataFrame` Method

```
createDataFrame(rdd: RDD[_], beanClass: Class[_]): DataFrame
createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame
createDataFrame[A <: Product : TypeTag](rdd: RDD[A]): DataFrame
createDataFrame[A <: Product : TypeTag](data: Seq[A]): DataFrame
// private[sql]
createDataFrame(rowRDD: RDD[Row], schema: StructType, needsConversion: Boolean): DataFrame
```

`createDataFrame` creates a `DataFrame` using `RDD[Row]` and the input `schema`. It is assumed that the rows in `rowRDD` all match the `schema`.

Caution

FIXME

Executing SQL Queries (aka SQL Mode) — `sql` Method

```
sql(sqlText: String): DataFrame
```

`sql` executes the `sqlText` SQL statement and creates a [DataFrame](#).

`sql` is imported in [spark-shell](#) so you can execute SQL statements as if `sql` were a part of the environment.

Note

```
scala> :imports
1) import spark.implicits._          (72 terms, 43 are implicit)
2) import spark.sql                 (1 terms)
```

```
scala> sql("SHOW TABLES")
res0: org.apache.spark.sql.DataFrame = [tableName: string, isTemporary: boolean]

scala> sql("DROP TABLE IF EXISTS testData")
res1: org.apache.spark.sql.DataFrame = []

// Let's create a table to SHOW it
spark.range(10).write.option("path", "/tmp/test").saveAsTable("testData")

scala> sql("SHOW TABLES").show
+-----+
|tableName|isTemporary|
+-----+-----+
| testdata|      false|
+-----+-----+
```

Internally, `sql` requests the current [ParserInterface](#) to execute a SQL query that gives a [LogicalPlan](#).

Note

`sql` uses `SessionState` to access the current [ParserInterface](#).

`sql` then creates a [DataFrame](#) using the current [SparkSession](#) (itself) and the [LogicalPlan](#).

[spark-sql](#) is the main SQL environment in Spark to work with pure SQL statements (where you do not have to use Scala to execute them).

Tip

```
spark-sql> show databases;
default
Time taken: 0.028 seconds, Fetched 1 row(s)
```

Accessing UDFRegistration — `udf` Attribute

```
udf: UDFRegistration
```

`udf` attribute gives access to [UDFRegistration](#) that allows registering user-defined functions for SQL-based queries.

```
val spark: SparkSession = ...
spark.udf.register("myUpper", (s: String) => s.toUpperCase)

val strs = ('a' to 'c').map(_.toString).toDS
strs.registerTempTable("strs")

scala> sql("SELECT *, myUpper(value) UPPER FROM strs").show
+-----+-----+
|value|UPPER|
+-----+-----+
|    a|    A|
|    b|    B|
|    c|    C|
+-----+-----+
```

Internally, it is simply an alias for [SessionState.udfRegistration](#).

Loading Data From Table — `table` Method

```
table(tableName: String): DataFrame (1)
// private[sql]
table(tableIdent: TableIdentifier): DataFrame
```

1. Parses `tableName` to a `TableIdentifier` and calls the other `table`

`table` creates a [DataFrame](#) (wrapper) from the input `tableName` table (but only if available in the session catalog).

```
scala> spark.catalog.tableExists("t1")
res1: Boolean = true

// t1 exists in the catalog
// let's load it
val t1 = spark.table("t1")
```

Accessing Metastore — `catalog` Attribute

```
catalog: Catalog
```

`catalog` attribute is a (lazy) interface to the current metastore, i.e. [data catalog](#) (of relational entities like databases, tables, functions, table columns, and views).

Tip

All methods in `catalog` return `Datasets`.

```
scala> spark.catalog.listTables.show
+-----+-----+-----+-----+
|       name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|my_permanent_table| default|      null|  MANAGED|     false|
|          strs|    null|      null|TEMPORARY|      true|
+-----+-----+-----+-----+
```

Internally, `catalog` creates a [CatalogImpl](#) (that uses the current `SparkSession`).

Accessing DataFrameReader — `read` method

```
read: DataFrameReader
```

`read` method returns a [DataFrameReader](#) that is used to read data from external storage systems and load it into a `DataFrame`.

```
val spark: SparkSession = // create instance
val dfReader: DataFrameReader = spark.read
```

Getting Runtime Configuration — `conf` Attribute

```
conf: RuntimeConfig
```

`conf` returns the current [RuntimeConfig](#).

Internally, `conf` creates a [RuntimeConfig](#) (when requested the very first time and cached afterwards) with the [SQLConf](#) of the [SessionState](#).

`readStream` method

```
readStream: DataStreamReader
```

`readStream` returns a new [DataStreamReader](#).

`streams` Attribute

```
streams: StreamingQueryManager
```

`streams` attribute gives access to [StreamingQueryManager](#) (through [SessionState](#)).

```
val spark: SparkSession = ...
spark.streams.active.foreach(println)
```

experimentalMethods Attribute

```
experimental: ExperimentalMethods
```

`experimentalMethods` is an extension point with [ExperimentalMethods](#) that is a per-session collection of extra strategies and `Rule[LogicalPlan]`s.

Note

`experimental` is used in [SparkPlanner](#) and [SparkOptimizer](#). Hive and [Structured Streaming](#) use it for their own extra strategies and optimization rules.

Creating SparkSession Instance — newSession method

```
newSession(): SparkSession
```

`newSession` creates (starts) a new `sparkSession` (with the current [SparkContext](#) and [SharedState](#)).

```
scala> val newSession = spark.newSession
newSession: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@122f
58a
```

Stopping SparkSession — stop Method

```
stop(): Unit
```

`stop` stops the `SparkSession`, i.e. [stops the underlying SparkContext](#).

Create DataFrame from BaseRelation — baseRelationToDataFrame Method

```
baseRelationToDataFrame(baseRelation: BaseRelation): DataFrame
```

Internally, `baseRelationToDataFrame` creates a `DataFrame` from the input `BaseRelation` wrapped inside `LogicalRelation`.

Note	<code>LogicalRelation</code> is an logical plan adapter for <code>BaseRelation</code> (so <code>BaseRelation</code> can be part of a <code>logical plan</code>).
------	---

Note	<code>baseRelationToDataFrame</code> is used when: <ul style="list-style-type: none"> • <code>DataFrameReader</code> loads data from a data source that supports multiple paths • <code>DataFrameReader</code> loads data from an external table using JDBC • <code>TextInputCSVDataSource</code> creates a base <code>Dataset</code> (of Strings) • <code>TextInputJsonDataSource</code> creates a base <code>Dataset</code> (of Strings)
------	--

Creating SessionState Instance — `instantiateSessionState` Internal Method

```
instantiateSessionState(className: String, sparkSession: SparkSession): SessionState
```

`instantiateSessionState` finds the `className` that is then used to `create` and `build` a `BaseSessionStateBuilder`.

`instantiateSessionState` may report an `IllegalArgumentException` while instantiating the class of a `SessionState`:

```
Error while instantiating '[className]'
```

Note	<code>instantiateSessionState</code> is used exclusively when <code>sparkSession</code> is requested for <code>SessionState</code> per <code>spark.sql.catalogImplementation</code> configuration property (and one is not available yet).
------	--

`sessionStateClassName` Internal Method

```
sessionStateClassName(conf: SparkConf): String
```

`sessionStateClassName` gives the name of the class of the `SessionState` per `spark.sql.catalogImplementation`, i.e.

- `org.apache.spark.sql.hive.HiveSessionStateBuilder` for `hive`
- `org.apache.spark.sql.internal.SessionStateBuilder` for `in-memory`

Note `sessionStateClassName` is used exclusively when `sparkSession` is requested for the `SessionState` (and one is not available yet).

Creating DataFrame From RDD Of Internal Binary Rows and Schema — `internalCreateDataFrame` Internal Method

```
internalCreateDataFrame(  
    catalystRows: RDD[InternalRow],  
    schema: StructType,  
    isStreaming: Boolean = false): DataFrame
```

`internalCreateDataFrame` creates a `DataFrame` with a `LogicalRDD`.

Note `internalCreateDataFrame` is used when:

- `DataFrameReader` is requested to create a `DataFrame` from Dataset of `JSONs or CSVs`
- `SparkSession` is requested to create a `DataFrame` from `RDD` of rows
- `InsertIntoDataSourceCommand` logical command is `executed`

Creating SparkSession Instance

`SparkSession` takes the following when created:

- Spark Core's `sparkContext`
- Optional `SharedState`
- Optional `SessionState`
- `SparkSessionExtensions`

`clearActiveSession` Object Method

```
clearActiveSession(): Unit
```

`clearActiveSession` ...FIXME

clearDefaultSession Object Method

```
clearDefaultSession(): Unit
```

```
clearDefaultSession ...FIXME
```

Accessing ExperimentalMethods — experimental Method

```
experimental: ExperimentalMethods
```

```
experimental ...FIXME
```

getActiveSession Object Method

```
getActiveSession: Option[SparkSession]
```

```
getActiveSession ...FIXME
```

getDefaultSession Object Method

```
getDefaultSession: Option[SparkSession]
```

```
getDefaultSession ...FIXME
```

Accessing ExecutionListenerManager — listenerManager Method

```
listenerManager: ExecutionListenerManager
```

```
listenerManager ...FIXME
```

Accessing SessionState — sessionState Lazy Attribute

```
sessionState: SessionState
```

```
sessionState ...FIXME
```

setActiveSession Object Method

```
setActiveSession(session: SparkSession): Unit
```

```
setActiveSession ...FIXME
```

setDefaultSession Object Method

```
setDefaultSession(session: SparkSession): Unit
```

```
setDefaultSession ...FIXME
```

Accessing SharedState — sharedState Method

```
sharedState: SharedState
```

```
sharedState ...FIXME
```

Measuring Duration of Executing Code Block — time Method

```
time[T](f: => T): T
```

```
time ...FIXME
```

Builder—Building SparkSession using Fluent API

`Builder` is the [fluent API](#) to create a `SparkSession`.

Table 1. Builder API

Method	Description
<code>appName</code>	<code>appName(name: String): Builder</code>
<code>config</code>	<code>config(conf: SparkConf): Builder</code> <code>config(key: String, value: Boolean): Builder</code> <code>config(key: String, value: Double): Builder</code> <code>config(key: String, value: Long): Builder</code> <code>config(key: String, value: String): Builder</code>
<code>enableHiveSupport</code>	Enables Hive support <code>enableHiveSupport(): Builder</code>
<code>getOrCreate</code>	Gets the current SparkSession or creates a new one. <code>getOrCreate(): SparkSession</code>
<code>master</code>	<code>master(master: String): Builder</code>
<code>withExtensions</code>	Access to the SparkSessionExtensions <code>withExtensions(f: SparkSessionExtensions => Unit): Builder</code>

`Builder` is available using the `builder` object method of a `SparkSession`.

```

import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder
  .appName("My Spark Application") // optional and will be autogenerated if not specified
  .master("local[*]")           // only for demo and testing purposes, use spark-submit instead
  .enableHiveSupport()          // self-explanatory, isn't it?
  .config("spark.sql.warehouse.dir", "target/spark-warehouse")
  .withExtensions { extensions =>
    extensions.injectResolutionRule { session =>
      ...
    }
    extensions.injectOptimizerRule { session =>
      ...
    }
  }
  .getOrCreate

```

Note You can have multiple `SparkSession`s in a single Spark application for different [data catalogs](#) (through relational entities).

Table 2. Builder's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>extensions</code>	SparkSessionExtensions Used when...FIXME
<code>options</code>	Used when...FIXME

Getting Or Creating SparkSession Instance — `getOrCreate` Method

```
getOrCreate(): SparkSession
```

```
getOrCreate ...FIXME
```

Enabling Hive Support — `enableHiveSupport` Method

```
enableHiveSupport(): Builder
```

`enableHiveSupport` enables Hive support, i.e. running structured queries on Hive tables (and a persistent Hive metastore, support for Hive serdes and Hive user-defined functions).

Note

You do **not** need any existing Hive installation to use Spark's Hive support. `SparkSession` context will automatically create `metastore_db` in the current directory of a Spark application and a directory configured by `spark.sql.warehouse.dir`.

Refer to [SharedState](#).

Internally, `enableHiveSupport` makes sure that the Hive classes are on CLASSPATH, i.e. Spark SQL's `org.apache.hadoop.hive.conf.HiveConf`, and sets `spark.sql.catalogImplementation` internal configuration property to `hive`.

withExtensions Method

```
withExtensions(f: SparkSessionExtensions => Unit): Builder
```

`withExtensions` simply executes the input `f` function with the [SparkSessionExtensions](#).

appName Method

```
appName(name: String): Builder
```

`appName` ...FIXME

config Method

```
config(conf: SparkConf): Builder
config(key: String, value: Boolean): Builder
config(key: String, value: Double): Builder
config(key: String, value: Long): Builder
config(key: String, value: String): Builder
```

`config` ...FIXME

master Method

```
master(master: String): Builder
```

`master` ...FIXME

implicits Object — Implicits Conversions

`implicits` object gives [implicit conversions](#) for converting Scala objects (incl. RDDs) into a `Dataset`, `DataFrame`, `Columns` or supporting such conversions (through [Encoders](#)).

Table 1. `implicits` API

Name	Description
<code>localSeqToDatasetHolder</code>	Creates a DatasetHolder with the input <code>Seq[T]</code> converted to a <code>Dataset</code> (<code>SparkSession.createDataset</code>). <pre>implicit def localSeqToDatasetHolder[T : Encoder](s: Seq[T]): Dataset[T] =</pre>
<code>Encoders</code>	Encoders for primitive and object types in Scala and Java (aka <code>Encoder</code> and <code>Decoder</code>)
<code>StringToColumn</code>	Converts <code>\$"name"</code> into a Column <pre>implicit class StringToColumn(val sc: StringContext):</pre>
<code>rddToDatasetHolder</code>	 <pre>implicit def rddToDatasetHolder[T : Encoder](rdd: RDD[T]): Dataset[T] =</pre>
<code>symbolToColumn</code>	 <pre>implicit def symbolToColumn(s: Symbol): ColumnName =</pre>

`implicits` object is defined inside [SparkSession](#) and hence requires that you build a [SparkSession](#) instance first before importing `implicits` conversions.

```

import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
import spark.implicits._

scala> val ds = Seq("I am a shiny Dataset!").toDS
ds: org.apache.spark.sql.Dataset[String] = [value: string]

scala> val df = Seq("I am an old grumpy DataFrame!").toDF
df: org.apache.spark.sql.DataFrame = [value: string]

scala> val df = Seq("I am an old grumpy DataFrame with text column!").toDF("text")
df: org.apache.spark.sql.DataFrame = [text: string]

val rdd = sc.parallelize(Seq("hello, I'm a very low-level RDD"))
scala> val ds = rdd.toDS
ds: org.apache.spark.sql.Dataset[String] = [value: string]

```

Tip In Scala REPL-based environments, e.g. `spark-shell`, use `:imports` to know what imports are in scope.

```

scala> :help imports

show import history, identifying sources of names

scala> :imports
1) import org.apache.spark.SparkContext._ (69 terms, 1 are implicit)
2) import spark.implicits._          (1 types, 67 terms, 37 are implicit)
3) import spark.sql                 (1 terms)
4) import org.apache.spark.sql.functions._ (354 terms)

```

`implicits` object extends `SQLImplicits` abstract class.

DatasetHolder Scala Case Class

`DatasetHolder` is a Scala case class that, when created, takes a `Dataset[T]`.

`DatasetHolder` is [created](#) (implicitly) when `rddToDatasetHolder` and `localSeqToDatasetHolder` implicit conversions are used.

`DatasetHolder` has `toDS` and `toDF` methods that simply return the `Dataset[T]` (it was created with) or a `DataFrame` (using `Dataset.toDF` operator), respectively.

```

toDS(): Dataset[T]
toDF(): DataFrame
toDF(colNames: String*): DataFrame

```


SparkSessionExtensions

`SparkSessionExtensions` is an [interface](#) that a Spark developer can use to extend a `SparkSession` with custom query execution rules and a relational entity parser.

As a Spark developer, you use [Builder.withExtensions](#) method (while building a new `SparkSession`) to access the session-bound `SparkSessionExtensions`.

Table 1. `SparkSessionExtensions` API

Method	Description
injectCheckRule	<code>injectCheckRule(builder: SparkSession => LogicalPlan =></code>
injectOptimizerRule	Registering a custom operator optimization rule <code>injectOptimizerRule(builder: SparkSession => Rule[LogicalOp] =></code>
injectParser	<code>injectParser(builder: (SparkSession, ParserInterface) =></code>
injectPlannerStrategy	<code>injectPlannerStrategy(builder: SparkSession => Strategy[LogicalPlan] =></code>
injectPostHocResolutionRule	<code>injectPostHocResolutionRule(builder: SparkSession => Rule[LogicalPlan] =></code>
injectResolutionRule	<code>injectResolutionRule(builder: SparkSession => Rule[LogicalPlan] =></code>

`SparkSessionExtensions` is an integral part of `SparkSession` (and is indirectly required to create one).

Table 2. SparkSessionExtensions's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
optimizerRules	<p>Collection of <code>RuleBuilder</code> functions (i.e. <code>SparkSession ⇒ Rule[LogicalPlan]</code>)</p> <p>Used when <code>SparkSessionExtensions</code> is requested to:</p> <ul style="list-style-type: none"> Associate custom operator optimization rules with <code>SparkSession</code> Register a custom operator optimization rule

Associating Custom Operator Optimization Rules with SparkSession — `buildOptimizerRules` Method

```
buildOptimizerRules(session: SparkSession): Seq[Rule[LogicalPlan]]
```

`buildOptimizerRules` gives the `optimizerRules` logical rules that are associated with the input `SparkSession`.

Note

`buildOptimizerRules` is used exclusively when `BaseSessionStateBuilder` is requested for the custom operator optimization rules to add to the base Operator Optimization batch.

Registering Custom Check Analysis Rule (Builder) — `injectCheckRule` Method

```
injectCheckRule(builder: SparkSession => LogicalPlan => Unit): Unit
```

`injectCheckRule ...FIXME`

Registering Custom Operator Optimization Rule (Builder) — `injectOptimizerRule` Method

```
injectOptimizerRule(builder: SparkSession => Rule[LogicalPlan]): Unit
```

`injectOptimizerRule` simply registers a custom operator optimization rule (as a `RuleBuilder` function) to the `optimizerRules` internal registry.

Registering Custom Parser (Builder) — `injectParser` Method

```
injectParser(builder: (SparkSession, ParserInterface) => ParserInterface): Unit
```

`injectParser` ...FIXME

Registering Custom Planner Strategy (Builder) — `injectPlannerStrategy` Method

```
injectPlannerStrategy(builder: SparkSession => Strategy): Unit
```

`injectPlannerStrategy` ...FIXME

Registering Custom Post-Hoc Resolution Rule (Builder) — `injectPostHocResolutionRule` Method

```
injectPostHocResolutionRule(builder: SparkSession => Rule[LogicalPlan]): Unit
```

`injectPostHocResolutionRule` ...FIXME

Registering Custom Resolution Rule (Builder) — `injectResolutionRule` Method

```
injectResolutionRule(builder: SparkSession => Rule[LogicalPlan]): Unit
```

`injectResolutionRule` ...FIXME

Dataset — Structured Query with Data Encoder

Dataset is a strongly-typed data structure in Spark SQL that represents a structured query.

Note

A structured query can be written using SQL or [Dataset API](#).

The following figure shows the relationship between different entities of Spark SQL that all together give the `Dataset` data structure.

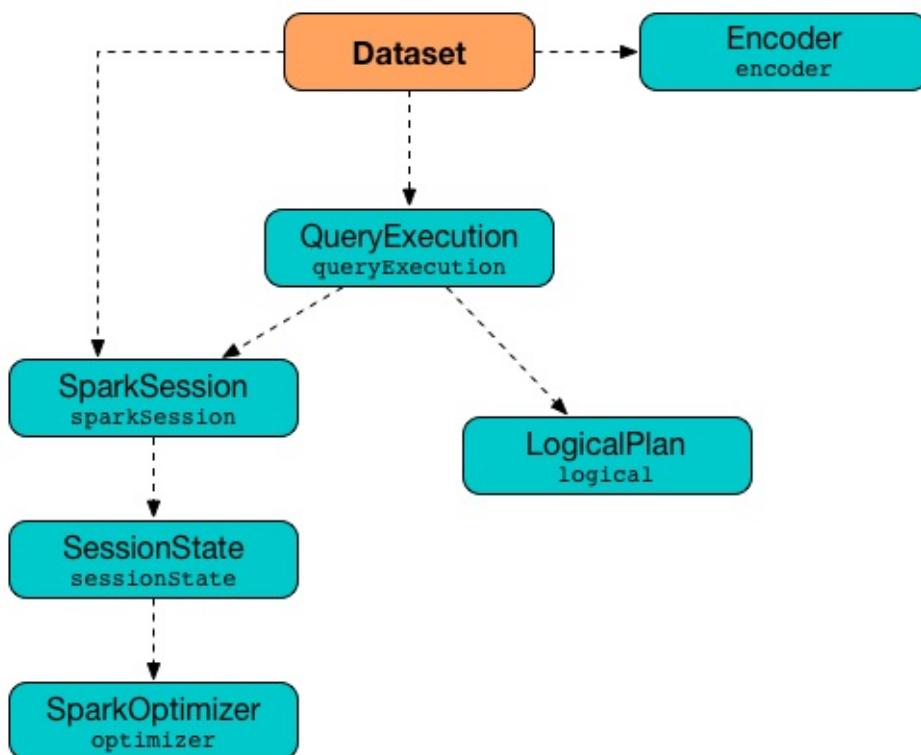


Figure 1. Dataset's Internals

It is therefore fair to say that `Dataset` consists of the following three elements:

1. [QueryExecution](#) (with the parsed unanalyzed [LogicalPlan](#) of a structured query)
2. [Encoder](#) (of the type of the records for fast serialization and deserialization to and from [InternalRow](#))
3. [SparkSession](#)

When [created](#), `Dataset` takes such a 3-element tuple with a `SparkSession`, a `QueryExecution` and an `Encoder`.

`Dataset` is [created](#) when:

- `Dataset.apply` (for a [LogicalPlan](#) and a [SparkSession](#) with the [Encoder](#) in a Scala implicit scope)

- `Dataset.ofRows` (for a `LogicalPlan` and a `SparkSession`)
- `Dataset.toDF` untyped transformation is used
- `Dataset.select`, `Dataset.randomSplit` and `Dataset.mapPartitions` typed transformations are used
- `KeyValueGroupedDataset.agg` operator is used (that requests `KeyValueGroupedDataset` to `aggUntyped`)
- `SparkSession.emptyDataset` and `SparkSession.range` operators are used
- `CatalogImpl` is requested to `makeDataset` (when requested to `list databases`, `tables`, `functions` and `columns`)
- Spark Structured Streaming's `MicroBatchExecution` is requested to `runBatch`

Datasets are *lazy* and structured query operators and expressions are only triggered when an action is invoked.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

scala> val dataset = spark.range(5)
dataset: org.apache.spark.sql.Dataset[Long] = [id: bigint]

// Variant 1: filter operator accepts a Scala function
dataset.filter(n => n % 2 == 0).count

// Variant 2: filter operator accepts a Column-based SQL expression
dataset.filter('value % 2 === 0).count

// Variant 3: filter operator accepts a SQL query
dataset.filter("value % 2 = 0").count
```

The [Dataset API](#) offers declarative and type-safe operators that makes for an improved experience for data processing (comparing to [DataFrames](#) that were a set of index- or column name-based [Rows](#)).

Note

`Dataset` was first introduced in Apache Spark **1.6.0** as an experimental feature, and has since turned itself into a fully supported API.

As of Spark **2.0.0**, [DataFrame](#) - the flagship data abstraction of previous versions of Spark SQL - is currently a *mere* type alias for `Dataset[Row]` :

```
type DataFrame = Dataset[Row]
```

See [package object sql](#).

Dataset offers convenience of RDDs with the performance optimizations of DataFrames and the strong static type-safety of Scala. The last feature of bringing the strong type-safety to DataFrame makes Dataset so appealing. All the features together give you a more functional programming interface to work with structured data.

```
scala> spark.range(1).filter('id === 0).explain(true)
== Parsed Logical Plan ==
'Filter ('id = 0)
+- Range (0, 1, splits=8)

== Analyzed Logical Plan ==
id: bigint
Filter (id#51L = cast(0 as bigint))
+- Range (0, 1, splits=8)

== Optimized Logical Plan ==
Filter (id#51L = 0)
+- Range (0, 1, splits=8)

== Physical Plan ==
*Filter (id#51L = 0)
+- *Range (0, 1, splits=8)

scala> spark.range(1).filter(_ == 0).explain(true)
== Parsed Logical Plan ==
'TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], un
resolveddeserializer(newInstance(class java.lang.Long))
+- Range (0, 1, splits=8)

== Analyzed Logical Plan ==
id: bigint
TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], new
Instance(class java.lang.Long)
+- Range (0, 1, splits=8)

== Optimized Logical Plan ==
TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], new
Instance(class java.lang.Long)
+- Range (0, 1, splits=8)

== Physical Plan ==
*Filter <function1>.apply
+- *Range (0, 1, splits=8)
```

It is only with Datasets to have syntax and analysis checks at compile time (that was not possible using DataFrame, regular SQL queries or even RDDs).

Using `Dataset` objects turns `DataFrames` of `Row` instances into a `DataFrames` of case classes with proper names and types (following their equivalents in the case classes). Instead of using indices to access respective fields in a `DataFrame` and cast it to a type, all this is automatically handled by Datasets and checked by the Scala compiler.

If however a `LogicalPlan` is used to create a `Dataset`, the logical plan is first executed (using the current `SessionState` in the `SparkSession`) that yields the `QueryExecution` plan.

A `Dataset` is `Queryable` and `Serializable`, i.e. can be saved to a persistent storage.

Note	<code>SparkSession</code> and <code>QueryExecution</code> are transient attributes of a <code>Dataset</code> and therefore do not participate in Dataset serialization. The only <i>firmly-tied</i> feature of a <code>Dataset</code> is the <code>Encoder</code> .
------	---

You can request the "untyped" view of a Dataset or access the `RDD` that is generated after executing the query. It is supposed to give you a more pleasant experience while transitioning from the legacy `RDD`-based or `DataFrame`-based APIs you may have used in the earlier versions of Spark SQL or encourage migrating from Spark Core's `RDD` API to Spark SQL's Dataset API.

The default storage level for `Datasets` is `MEMORY_AND_DISK` because recomputing the in-memory columnar representation of the underlying table is expensive. You can however persist a `Dataset`.

Note	Spark 2.0 has introduced a new query model called <code>Structured Streaming</code> for continuous incremental execution of structured queries. That made possible to consider Datasets a static and bounded as well as streaming and unbounded data sets with a single unified API for different execution models.
------	---

A `Dataset` is `local` if it was created from local collections using `SparkSession.emptyDataset` or `SparkSession.createDataset` methods and their derivatives like `toDF`. If so, the queries on the Dataset can be optimized and run locally, i.e. without using Spark executors.

Note	<code>Dataset</code> makes sure that the underlying <code>queryExecution</code> is <code>analyzed</code> and <code>checked</code> .
------	---

Table 1. Dataset's Properties

Name	Description
boundEnc	<code>ExpressionEncoder</code> Used when...FIXME
	Deserializer <code>expression</code> to convert internal rows to objects of type <code>T</code> Created lazily by requesting the <code>ExpressionEncoder</code> to <code>resolveAndBind</code> Used when:

deserializer	<ul style="list-style-type: none"> • Dataset is created (for a logical plan in a given sparkSession) • <code>Dataset.toLocalIterator</code> operator is used (to create a Java Iterator type <code>T</code>) • Dataset is requested to collect all rows from a spark plan
exprEnc	<p>Implicit ExpressionEncoder</p> <p>Used when...FIXME</p>
logicalPlan	<p>Analyzed logical plan with all logical commands executed and turned LocalRelation.</p> <p><code>logicalPlan: LogicalPlan</code></p> <p>When initialized, <code>logicalPlan</code> requests the QueryExecution for analysis; if the plan is a logical command or a union thereof, <code>logicalPlan</code> executes the QueryExecution (using <code>executeCollect</code>).</p>
planWithBarrier	<p><code>planWithBarrier: AnalysisBarrier</code></p>
rdd	<p>(lazily-created) RDD of JVM objects of type <code>T</code> (as converted from rows in the internal binary row format).</p> <p><code>rdd: RDD[T]</code></p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>Note</p> <p><code>rdd</code> gives <code>RDD</code> with the extra execution step to convert rows in the internal binary row format to JVM objects that will impact the performance of the code as the objects are inside JVM (while were outside before). It's better to use <code>rdd</code> directly.</p> </div> <p>Internally, <code>rdd</code> first creates a new logical plan that deserializes the Data Encoding Plan.</p> <pre> val dataset = spark.range(5).withColumn("group", 'id % 2) scala> dataset.rdd.toDebugString res1: String = (8) MapPartitionsRDD[8] at rdd at <console>:26 [] MapPartitionsRDD[7] at rdd at <console>:26 [] MapPartitionsRDD[6] at rdd at <console>:26 [] MapPartitionsRDD[5] at rdd at <console>:26 [] ParallelCollectionRDD[4] at rdd at <console>:26 [] scala> dataset.queryExecution.toRdd.toDebugString res2: String = (8) MapPartitionsRDD[11] at toRdd at <console>:26 [] MapPartitionsRDD[10] at toRdd at <console>:26 [] ParallelCollectionRDD[9] at toRdd at <console>:26 [] </pre>

	<code>rdd</code> then requests <code>SessionState</code> to execute the logical plan to get the RDD of binary rows .
Note	<code>rdd</code> uses <code>SparkSession</code> to access <code>SessionState</code> .
	<code>rdd</code> then requests the Dataset's <code>ExpressionEncoder</code> for the <code>data type</code> <code>deserializer</code> expression) and maps over them (per partition) to create the expected type <code>T</code> .
Note	<code>rdd</code> is at the "boundary" between the internal binary row format and the JVM type of the dataset. Avoid the extra deserialization step to reduce memory requirements of your Spark application.
<code>sqlContext</code>	Lazily-created <code>SQLContext</code> Used when...FIXME

Getting Input Files of Relations (in Structured Query)

— `inputFiles` Method

```
inputFiles: Array[String]
```

`inputFiles` requests `QueryExecution` for optimized logical plan and collects the following logical operators:

- `LogicalRelation` with `FileRelation` (as the `BaseRelation`)
- `FileRelation`
- `HiveTableRelation`

`inputFiles` then requests the logical operators for their underlying files:

- `inputFiles` of the `FileRelations`
- `locationUri` of the `HiveTableRelation`

resolve Internal Method

```
resolve(colName: String): NamedExpression
```

Caution	FIXME
---------	-------

Creating Dataset Instance

`Dataset` takes the following when created:

- `SparkSession`
- `QueryExecution`
- `Encoder` for the type `T` of the records

Note

You can also create a `Dataset` using `LogicalPlan` that is immediately `executed` using `SessionState`.

Internally, `Dataset` requests `QueryExecution` to `analyze` itself.

`Dataset` initializes the `internal registries and counters`.

Is Dataset Local? — `isLocal` Method

```
isLocal: Boolean
```

`isLocal` flag is enabled (i.e. `true`) when operators like `collect` or `take` could be run locally, i.e. without using executors.

Internally, `isLocal` checks whether the logical query plan of a `Dataset` is `LocalRelation`.

Is Dataset Streaming? — `isStreaming` method

```
isStreaming: Boolean
```

`isStreaming` is enabled (i.e. `true`) when the logical plan `is streaming`.

Internally, `isStreaming` takes the Dataset's `logical plan` and gives `whether the plan is streaming or not`.

Queryable

Caution**FIXME**

`withNewRDDExecutionId` Internal Method

```
withNewRDDExecutionId[U](body: => U): U
```

`withNewRDDExecutionId` executes the input `body` action under `new execution id`.

Caution	FIXME What's the difference between <code>withNewRDDExecutionId</code> and <code>withNewExecutionId</code> ?
Note	<code>withNewRDDExecutionId</code> is used when <code>Dataset.foreach</code> and <code>Dataset.foreachPartition</code> actions are used.

Creating DataFrame (For Logical Query Plan and SparkSession) — `ofRows` Internal Factory Method

```
ofRows(sparkSession: SparkSession, logicalPlan: LogicalPlan): DataFrame
```

Note	<code>ofRows</code> is part of <code>Dataset</code> Scala object that is marked as a <code>private[sql]</code> and so can only be accessed from code in <code>org.apache.spark.sql</code> package.
------	--

`ofRows` returns `DataFrame` (which is the type alias for `Dataset[Row]`). `ofRows` uses `RowEncoder` to convert the schema (based on the input `logicalPlan` logical plan).

Internally, `ofRows` prepares the input `logicalPlan` for execution and creates a `Dataset[Row]` with the current `SparkSession`, the `QueryExecution` and `RowEncoder`.

Note	<p><code>ofRows</code> is used when:</p> <ul style="list-style-type: none"> • <code>DataFrameReader</code> is requested to load data from a data source • <code>Dataset</code> is requested to execute <code>checkpoint</code>, <code>mapPartitionsInR</code>, <code>untyped transformations</code> and <code>set-based typed transformations</code> • <code>RelationalGroupedDataset</code> is requested to create a <code>DataFrame</code> from <code>aggregate expressions</code>, <code>flatMapGroupsInR</code> and <code>flatMapGroupsInPandas</code> • <code>SparkSession</code> is requested to create a <code>DataFrame</code> from a <code>BaseRelation</code>, <code>createDataFrame</code>, <code>internalCreateDataFrame</code>, <code>sql</code> and <code>table</code> • <code>CacheTableCommand</code>, <code>CreateTempViewUsing</code>, <code>InsertIntoDataSourceCommand</code> and <code>SaveIntoDataSourceCommand</code> logical commands are executed (run) • <code>DataSource</code> is requested to <code>writeAndRead</code> (for a <code>CreatableRelationProvider</code>) • <code>FrequentItems</code> is requested to <code>singlePassFreqItems</code> • <code>StatFunctions</code> is requested to <code>crossTabulate</code> and <code>summary</code> • Spark Structured Streaming's <code>DatastreamReader</code> is requested to <code>load</code> • Spark Structured Streaming's <code>DataStreamWriter</code> is requested to <code>start</code> • Spark Structured Streaming's <code>FileStreamSource</code> is requested to <code>getBatch</code> • Spark Structured Streaming's <code>MemoryStream</code> is requested to <code>toDF</code>
------	---

Tracking Multi-Job Structured Query Execution (PySpark)

— `withNewExecutionId` Internal Method

```
withNewExecutionId[U](body: => U): U
```

`withNewExecutionId` executes the input `body` action under new execution id.

Note	<code>withNewExecutionId</code> sets a unique execution id so that all Spark jobs belong to the <code>Dataset</code> action execution.
------	--

Note	<code>withNewExecutionId</code> is used exclusively when <code>Dataset</code> is executing Python-based actions (i.e. <code>collectToPython</code> , <code>collectAsArrowToPython</code> and <code>toPythonIterator</code>) that are not of much interest in this gitbook.
------	---

Feel free to contact me at jacek@japila.pl if you think I should re-consider my decision.

Executing Action Under New Execution ID

— `withAction` Internal Method

```
withAction[U](name: String, qe: QueryExecution)(action: SparkPlan => U)
```

`withAction` requests `QueryExecution` for the optimized physical query plan and resets the metrics of every physical operator (in the physical plan).

`withAction` requests `SQLExecution` to execute the input `action` with the executable physical plan (tracked under a new execution id).

In the end, `withAction` notifies `ExecutionListenerManager` that the `name` action has finished successfully or with an exception.

Note	<code>withAction</code> uses <code>SparkSession</code> to access <code>ExecutionListenerManager</code> .
------	--

Note	<code>withAction</code> is used when <code>Dataset</code> is requested for the following: <ul style="list-style-type: none"> • Computing the logical plan (and executing a logical command or their Union) • Dataset operators: <code>collect</code>, <code>count</code>, <code>head</code> and <code>toLocalIterator</code>
------	---

Creating Dataset Instance (For LogicalPlan and SparkSession) — `apply` Internal Factory Method

```
apply[T: Encoder](sparkSession: SparkSession, logicalPlan: LogicalPlan): Dataset[T]
```

Note	<code>apply</code> is part of <code>Dataset</code> Scala object that is marked as a <code>private[sql]</code> and so can only be accessed from code in <code>org.apache.spark.sql</code> package.
------	---

`apply` ...FIXME

Note	<code>apply</code> is used when: <ul style="list-style-type: none"> • <code>Dataset</code> is requested to execute typed transformations and set-based typed transformations • Spark Structured Streaming's <code>MemoryStream</code> is requested to <code>toDS</code>
------	---

Collecting All Rows From Spark Plan — `collectFromPlan` Internal Method

```
collectFromPlan(plan: SparkPlan): Array[T]
```

`collectFromPlan` ...FIXME

Note	<code>collectFromPlan</code> is used for Dataset.head , Dataset.collect and Dataset.collectAsList operators.
------	--

`selectUntyped` Internal Method

```
selectUntyped(columns: TypedColumn[_, _]*): Dataset[_]
```

`selectUntyped` ...FIXME

Note	<code>selectUntyped</code> is used exclusively when Dataset.select typed transformation is used.
------	--

Helper Method for Typed Transformations — `withTypedPlan` Internal Method

```
withTypedPlan[U: Encoder](logicalPlan: LogicalPlan): Dataset[U]
```

`withTypedPlan` ...FIXME

Note

`withTypedPlan` is annotated with Scala's `@inline` annotation that requests the Scala compiler to try especially hard to inline it.

Note

`withTypedPlan` is used in the `Dataset` typed transformations, i.e. `withWatermark`, `joinWith`, `hint`, `as`, `filter`, `limit`, `sample`, `dropDuplicates`, `filter`, `map`, `repartition`, `repartitionByRange`, `coalesce` and `sort` with `sortWithinPartitions` (through the `sortInternal` internal method).

Helper Method for Set-Based Typed Transformations

— `withSetOperator` Internal Method

```
withSetOperator[U: Encoder](logicalPlan: LogicalPlan): Dataset[U]
```

`withSetOperator` ...FIXME

Note

`withSetOperator` is annotated with Scala's `@inline` annotation that requests the Scala compiler to try especially hard to inline it.

Note

`withSetOperator` is used in the `Dataset` typed transformations, i.e. `union`, `unionByName`, `intersect` and `except`.

sortInternal Internal Method

```
sortInternal(global: Boolean, sortExprs: Seq[Column]): Dataset[T]
```

`sortInternal` creates a `Dataset` with `Sort` unary logical operator (and the `logicalPlan` as the child logical plan).

```
val nums = Seq((0, "zero"), (1, "one")).toDF("id", "name")
// Creates a Sort logical operator:
// - descending sort direction for id column (specified explicitly)
// - name column is wrapped with ascending sort direction
val numsSorted = nums.sort('id.desc, 'name)
val logicalPlan = numsSorted.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Sort ['id DESC NULLS LAST, 'name ASC NULLS FIRST], true
01 +- Project [_1#11 AS id#14, _2#12 AS name#15]
02     +- LocalRelation [_1#11, _2#12]
```

Internally, `sortInternal` firstly builds ordering expressions for the given `sortExprs` columns, i.e. takes the `sortExprs` columns and makes sure that they are `SortOrder` expressions already (and leaves them untouched) or wraps them into `SortOrder` expressions

with [Ascending](#) sort direction.

In the end, `sortInternal` creates a Dataset with `Sort` unary logical operator (with the ordering expressions, the given `global` flag, and the `logicalPlan` as the child logical plan).

Note	<code>sortInternal</code> is used for the <code>sort</code> and <code>sortWithinPartitions</code> typed transformations in the Dataset API (with the only change of the <code>global</code> flag being enabled and disabled, respectively).
------	---

Helper Method for Untyped Transformations and Basic Actions — `withPlan` Internal Method

<code>withPlan(logicalPlan: LogicalPlan): DataFrame</code>
--

`withPlan` simply uses [ofRows](#) internal factory method to create a `DataFrame` for the input `LogicalPlan` and the current [SparkSession](#).

Note	<code>withPlan</code> is annotated with Scala's @inline annotation that requests the Scala compiler to try especially hard to inline it.
------	--

Note	<code>withPlan</code> is used in the <code>dataset</code> untyped transformations (i.e. <code>join</code> , <code>crossJoin</code> and <code>select</code>) and basic actions (i.e. <code>createTempView</code> , <code>createOrReplaceTempView</code> , <code>createGlobalTempView</code> and <code>createOrReplaceGlobalTempView</code>).
------	---

Further Reading and Watching

- (video) [Structuring Spark: DataFrames, Datasets, and Streaming](#)

DataFrame — Dataset of Rows with RowEncoder

Spark SQL introduces a tabular functional data abstraction called **DataFrame**. It is designed to ease developing Spark applications for processing large amount of structured tabular data on Spark infrastructure.

DataFrame is a data abstraction or a domain-specific language (DSL) for working with **structured** and **semi-structured data**, i.e. datasets that you can specify a schema for.

DataFrame is a collection of **rows** with a **schema** that is the result of executing a structured query (once it will have been executed).

DataFrame uses the immutable, in-memory, resilient, distributed and parallel capabilities of **RDD**, and applies a structure called schema to the data.

Note

In **Spark 2.0.0** `DataFrame` is a *mere* type alias for `Dataset[Row]`.

```
type DataFrame = Dataset[Row]
```

See [org.apache.spark.package.scala](#).

`DataFrame` is a distributed collection of tabular data organized into **rows** and **named columns**. It is conceptually equivalent to a table in a relational database with operations to project (`select`), `filter`, `intersect`, `join`, `group`, `sort`, `join`, `aggregate`, or convert to a RDD (consult [DataFrame API](#))

```
data.groupBy('Product_ID).sum('Score)
```

Spark SQL borrowed the concept of DataFrame from [pandas' DataFrame](#) and made it **immutable**, **parallel** (one machine, perhaps with many processors and cores) and **distributed** (many machines, perhaps with many processors and cores).

Note

Hey, big data consultants, time to help teams migrate the code from pandas' DataFrame into Spark's DataFrames (at least to PySpark's DataFrame) and offer services to set up large clusters!

DataFrames in Spark SQL strongly rely on [the features of RDD](#) - it's basically a RDD exposed as structured DataFrame by appropriate operations to handle very big data from the day one. So, petabytes of data should *not* scare you (unless you're an administrator to create such clustered Spark environment - [contact me when you feel alone with the task](#)).

```

val df = Seq(("one", 1), ("one", 1), ("two", 1))
    .toDF("word", "count")

scala> df.show
+---+---+
|word|count|
+---+---+
| one|    1|
| one|    1|
| two|    1|
+---+---+

val counted = df.groupBy('word).count

scala> counted.show
+---+---+
|word|count|
+---+---+
| two|    1|
| one|    2|
+---+---+

```

You can create DataFrames by [loading data from structured files \(JSON, Parquet, CSV\)](#), [RDDs, tables in Hive, or external databases \(JDBC\)](#). You can also create DataFrames from scratch and build upon them (as in the above example). See [DataFrame API](#). You can read any format given you have appropriate Spark SQL extension of [DataFrameReader](#) to format the dataset appropriately.

Caution	FIXME Diagram of reading data from sources to create DataFrame
---------	--

You can execute queries over DataFrames using two approaches:

- [the good ol' SQL](#) - helps migrating from "SQL databases" world into the world of DataFrame in Spark SQL
- [Query DSL](#) - an API that helps ensuring proper syntax at compile time.

`DataFrame` also allows you to do the following tasks:

- [Filtering](#)

DataFrames use the [Catalyst query optimizer](#) to produce efficient queries (and so they are supposed to be faster than corresponding RDD-based queries).

Note	Your DataFrames can also be type-safe and moreover further improve their performance through specialized encoders that can significantly cut serialization and deserialization times.
------	---

You can enforce types on [generic rows](#) and hence bring type safety (at compile time) by [encoding rows into type-safe `dataset` object](#). As of Spark 2.0 it is a preferred way of developing Spark applications.

Features of DataFrame

A `DataFrame` is a collection of "generic" `Row` instances (as `RDD[Row]`) and a [schema](#).

Note

Regardless of how you create a `DataFrame`, it will always be a pair of `RDD[Row]` and [StructType](#).

SQLContext, spark, and Spark shell

You use [org.apache.spark.sql.SQLContext](#) to build DataFrames and execute SQL queries.

The quickest and easiest way to work with Spark SQL is to use [Spark shell](#) and `spark` object.

```
scala> spark
res1: org.apache.spark.sql.SQLContext = org.apache.spark.sql.hive.HiveContext@60ae950f
```

As you may have noticed, `spark` in Spark shell is actually a [org.apache.spark.sql.hive.HiveContext](#) that integrates **the Spark SQL execution engine** with data stored in [Apache Hive](#).

The Apache Hive™ data warehouse software facilitates querying and managing large datasets residing in distributed storage.

Creating DataFrames from Scratch

Use Spark shell as described in [Spark shell](#).

Using toDF

After you `import spark.implicits._` (which is done for you by Spark shell) you may apply `toDF` method to convert objects to DataFrames.

```
scala> val df = Seq("I am a DataFrame!").toDF("text")
df: org.apache.spark.sql.DataFrame = [text: string]
```

Creating DataFrame using Case Classes in Scala

This method assumes the data comes from a Scala case class that will describe the schema.

```
scala> case class Person(name: String, age: Int)
defined class Person

scala> val people = Seq(Person("Jacek", 42), Person("Patryk", 19), Person("Maksym", 5))
people: Seq[Person] = List(Person(Jacek,42), Person(Patryk,19), Person(Maksym,5))

scala> val df = spark.createDataFrame(people)
df: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> df.show
+---+---+
| name|age|
+---+---+
| Jacek| 42|
| Patryk| 19|
| Maksym| 5|
+---+---+
```

Custom DataFrame Creation using `createDataFrame`

`SQLContext` offers a family of `createDataFrame` operations.

```
scala> val lines = sc.textFile("Cartier+for+WinnersCurse.csv")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>
:24

scala> val headers = lines.first
headers: String = auctionid,bid,bidtime,bidder,bidderrate,openbid,price

scala> import org.apache.spark.sql.types.{StructField, StringType}
import org.apache.spark.sql.types.{StructField, StringType}

scala> val fs = headers.split(",").map(f => StructField(f, StringType))
fs: Array[org.apache.spark.sql.types.StructField] = Array(StructField(auctionid, StringType, true), StructField(bid, StringType, true), StructField(bidtime, StringType, true), StructField(bidder, StringType, true), StructField(bidderrate, StringType, true), StructField(openbid, StringType, true), StructField(price, StringType, true))

scala> import org.apache.spark.sql.types.StructType
import org.apache.spark.sql.types.StructType

scala> val schema = StructType(fs)
schema: org.apache.spark.sql.types.StructType = StructType(StructField(auctionid, StringType, true), StructField(bid, StringType, true), StructField(bidtime, StringType, true), StructField(bidder, StringType, true), StructField(bidderrate, StringType, true), StructField(openbid, StringType, true), StructField(price, StringType, true))
```

```

scala> val noheaders = lines.filter(_ != header)
noheaders: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at filter at <console>:33

scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row

scala> val rows = noheaders.map(_.split(",")).map(a => Row.fromSeq(a))
rows: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[12] at map at <console>:35

scala> val auctions = spark.createDataFrame(rows, schema)
auctions: org.apache.spark.sql.DataFrame = [auctionid: string, bid: string, bidtime: string, bidder: string, bidderrate: string, openbid: string, price: string]

scala> auctions.printSchema
root
|-- auctionid: string (nullable = true)
|-- bid: string (nullable = true)
|-- bidtime: string (nullable = true)
|-- bidder: string (nullable = true)
|-- bidderrate: string (nullable = true)
|-- openbid: string (nullable = true)
|-- price: string (nullable = true)

scala> auctions.dtypes
res28: Array[(String, String)] = Array((auctionid,StringType), (bid,StringType), (bidtime,StringType), (bidder,StringType), (bidderrate,StringType), (openbid,StringType), (price,StringType))

scala> auctions.show(5)
+-----+-----+-----+-----+-----+
| auctionid| bid| bidtime| bidder| bidderrate| openbid| price|
+-----+-----+-----+-----+-----+
|1638843936| 500|0.478368056| kona-java|      181|     500| 1625|
|1638843936| 800|0.826388889| doc213|       60|     500| 1625|
|1638843936| 600|3.761122685| zmxu|        7|     500| 1625|
|1638843936|1500|5.226377315|carloss8055|        5|     500| 1625|
|1638843936|1600| 6.570625| jdrinaz|        6|     500| 1625|
+-----+-----+-----+-----+-----+
only showing top 5 rows

```

Loading data from structured files

Creating DataFrame from CSV file

Let's start with an example in which **schema inference** relies on a custom case class in Scala.

```

scala> val lines = sc.textFile("Cartier+for+WinnersCurse.csv")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>
:24

scala> val header = lines.first
header: String = auctionid,bid,bidtime,bidder,bidderrate,openbid,price

scala> lines.count
res3: Long = 1349

scala> case class Auction(auctionid: String, bid: Float, bidtime: Float, bidder: String, bidderrate: Int, openbid: Float, price: Float)
defined class Auction

scala> val noheader = lines.filter(_ != header)
noheader: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[53] at filter at <console>:31

scala> val auctions = noheader.map(_.split(",")).map(r => Auction(r(0), r(1).toFloat,
r(2).toFloat, r(3), r(4).toInt, r(5).toFloat, r(6).toFloat))
auctions: org.apache.spark.rdd.RDD[Auction] = MapPartitionsRDD[59] at map at <console>:35

scala> val df = auctions.toDF
df: org.apache.spark.sql.DataFrame = [auctionid: string, bid: float, bidtime: float, bidder: string, bidderrate: int, openbid: float, price: float]

scala> df.printSchema
root
|-- auctionid: string (nullable = true)
|-- bid: float (nullable = false)
|-- bidtime: float (nullable = false)
|-- bidder: string (nullable = true)
|-- bidderrate: integer (nullable = false)
|-- openbid: float (nullable = false)
|-- price: float (nullable = false)

scala> df.show
+-----+-----+-----+-----+-----+-----+
| auctionid|    bid|   bidtime|      bidder|bidderrate|openbid|  price|
+-----+-----+-----+-----+-----+-----+
|1638843936| 500.0|0.47836804|  kona-java|       181| 500.0|1625.0|
|1638843936| 800.0| 0.8263889| doc213|        60| 500.0|1625.0|
|1638843936| 600.0| 3.7611227|      zmxu|         7| 500.0|1625.0|
|1638843936|1500.0| 5.2263775| carloss8055|        5| 500.0|1625.0|
|1638843936|1600.0| 6.570625| jdrinaz|        6| 500.0|1625.0|
|1638843936|1550.0| 6.8929167| carloss8055|        5| 500.0|1625.0|
|1638843936|1625.0| 6.8931136| carloss8055|        5| 500.0|1625.0|
|1638844284| 225.0| 1.237419|dre_313@yahoo.com|          0| 200.0| 500.0|
|1638844284| 500.0| 1.2524074| njbirdmom|        33| 200.0| 500.0|
|1638844464| 300.0| 1.8111342|  aprefer|        58| 300.0| 740.0|
|1638844464| 305.0| 3.2126737| 197509260|         3| 300.0| 740.0|

```

```
|1638844464| 450.0| 4.1657987|      coharley|      30| 300.0| 740.0|
|1638844464| 450.0| 6.7363195|      adammurry|       5| 300.0| 740.0|
|1638844464| 500.0| 6.7364697|      adammurry|       5| 300.0| 740.0|
|1638844464| 505.78| 6.9881945| 197509260|       3| 300.0| 740.0|
|1638844464| 551.0| 6.9896526| 197509260|       3| 300.0| 740.0|
|1638844464| 570.0| 6.9931483| 197509260|       3| 300.0| 740.0|
|1638844464| 601.0| 6.9939003| 197509260|       3| 300.0| 740.0|
|1638844464| 610.0| 6.994965| 197509260|       3| 300.0| 740.0|
|1638844464| 560.0| 6.9953704|      ps138|       5| 300.0| 740.0|
+-----+-----+-----+-----+-----+-----+
only showing top 20 rows
```

Creating DataFrame from CSV files using spark-csv module

You're going to use [spark-csv](#) module to load data from a CSV data source that handles proper parsing and loading.

Note

Support for CSV data sources is available by default in Spark 2.0.0. No need for an external module.

Start the Spark shell using `--packages` option as follows:

```

→ spark git:(master) ✘ ./bin/spark-shell --packages com.databricks:spark-csv_2.11:1.2
.0
Ivy Default Cache set to: /Users/jacek/.ivy2/cache
The jars for the packages stored in: /Users/jacek/.ivy2/jars
:: loading settings :: url = jar:file:/Users/jacek/dev/oss/spark/assembly/target/scala-
-2.11/spark-assembly-1.5.0-SNAPSHOT-hadoop2.7.1.jar!/org/apache/ivy/core/settings/ivys
ettings.xml
com.databricks#spark-csv_2.11 added as a dependency

scala> val df = spark.read.format("com.databricks.spark.csv").option("header", "true")
.load("Cartier+for+WinnersCurse.csv")
df: org.apache.spark.sql.DataFrame = [auctionid: string, bid: string, bidtime: string,
bidder: string, bidderrate: string, openbid: string, price: string]

scala> df.printSchema
root
|-- auctionid: string (nullable = true)
|-- bid: string (nullable = true)
|-- bidtime: string (nullable = true)
|-- bidder: string (nullable = true)
|-- bidderrate: string (nullable = true)
|-- openbid: string (nullable = true)
|-- price: string (nullable = true)

scala> df.show
+-----+-----+-----+-----+-----+-----+
| auctionid|    bid|   bidtime|      bidder|bidderrate|openbid|price|
+-----+-----+-----+-----+-----+-----+
|1638843936|  500|0.478368056|  kona-java|       181|    500| 1625|
|1638843936|  800|0.826388889| doc213|        60|    500| 1625|
|1638843936|  600|3.761122685|      zmxu|         7|    500| 1625|
|1638843936| 1500|5.226377315| carloss8055|        5|    500| 1625|
|1638843936| 1600| 6.570625| jdrinaz|        6|    500| 1625|
|1638843936| 1550|6.892916667| carloss8055|        5|    500| 1625|
|1638843936| 1625|6.893113426| carloss8055|        5|    500| 1625|
|1638844284|  225|1.237418982|dre_313@yahoo.com|        0|    200| 500|
|1638844284|  500|1.252407407| njbirdmom|       33|    200| 500|
|1638844464|  300|1.811134259| aprefer|       58|    300| 740|
|1638844464|  305|3.212673611| 197509260|        3|    300| 740|
|1638844464|  450|4.165798611| coharley|       30|    300| 740|
|1638844464|  450|6.736319444| adamurry|        5|    300| 740|
|1638844464|  500|6.736469907| adamurry|        5|    300| 740|
|1638844464| 505.78|6.988194444| 197509260|        3|    300| 740|
|1638844464|  551|6.989652778| 197509260|        3|    300| 740|
|1638844464|  570|6.993148148| 197509260|        3|    300| 740|
|1638844464|  601|6.993900463| 197509260|        3|    300| 740|
|1638844464|  610|6.994965278| 197509260|        3|    300| 740|
|1638844464|  560| 6.99537037| ps138|        5|    300| 740|
+-----+-----+-----+-----+-----+-----+
only showing top 20 rows

```

Reading Data from External Data Sources (read method)

You can create DataFrames by loading data from structured files (JSON, Parquet, CSV), RDDs, tables in Hive, or external databases (JDBC) using `SQLContext.read` method.

```
read: DataFrameReader
```

`read` returns a `DataFrameReader` instance.

Among the supported structured data (file) formats are (consult [Specifying Data Format \(format method\)](#) for `DataFrameReader`):

- JSON
- parquet
- JDBC
- ORC
- Tables in Hive and any JDBC-compliant database
- libsvm

```
val reader = spark.read
r: org.apache.spark.sql.DataFrameReader = org.apache.spark.sql.DataFrameReader@59e67a1
8

reader.parquet("file.parquet")
reader.json("file.json")
reader.format("libsvm").load("sample_libsvm_data.txt")
```

Querying DataFrame

Note

Spark SQL offers a [Pandas-like Query DSL](#).

Using Query DSL

You can select specific columns using `select` method.

Note

This variant (in which you use stringified column names) can only select existing columns, i.e. you cannot create new ones using select expressions.

```
scala> predictions.printSchema
root
|-- id: long (nullable = false)
|-- topic: string (nullable = true)
|-- text: string (nullable = true)
|-- label: double (nullable = true)
|-- words: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- features: vector (nullable = true)
|-- rawPrediction: vector (nullable = true)
|-- probability: vector (nullable = true)
|-- prediction: double (nullable = true)

scala> predictions.select("label", "words").show
+-----+-----+
|label|      words|
+-----+-----+
| 1.0|[hello, math!]|
| 0.0|[hello, religion!]|
| 1.0|[hello, phy, ic, !]|
+-----+-----+
```

```
scala> auctions.groupBy("bidder").count().show(5)
+-----+-----+
|      bidder|count|
+-----+-----+
|dennisthemenace1|    1|
|amskymom|      5|
|nguyenat@san.rr.com|    4|
|millyjohn|    1|
|ykelectro@hotmail...|    2|
+-----+-----+
only showing top 5 rows
```

In the following example you query for the top 5 of the most active bidders.

Note the `tiny` `$` and `desc` together with the column name to sort the rows by.

```
scala> auctions.groupBy("bidder").count().sort($"count".desc).show(5)
+-----+----+
|    bidder|count|
+-----+----+
|    lass1004|   22|
|  pascal1666|   19|
|    freembd|   17|
|restdynamics|   17|
|  happyrova|   17|
+-----+----+
only showing top 5 rows

scala> import org.apache.spark.sql.functions._  
import org.apache.spark.sql.functions._

scala> auctions.groupBy("bidder").count().sort(desc("count")).show(5)
+-----+----+
|    bidder|count|
+-----+----+
|    lass1004|   22|
|  pascal1666|   19|
|    freembd|   17|
|restdynamics|   17|
|  happyrova|   17|
+-----+----+
only showing top 5 rows
```

```

scala> df.select("auctionid").distinct.count
res88: Long = 97

scala> df.groupBy("bidder").count.show
+-----+----+
|      bidder | count |
+-----+----+
| dennisisthemenace1 | 1 |
| amskymom | 5 |
| nguyenat@san.rr.com | 4 |
| millyjohn | 1 |
| ykelectro@hotmail... | 2 |
| shetellia@aol.com | 1 |
| rrolex | 1 |
| bupper99 | 2 |
| cheddaboy | 2 |
| adcc007 | 1 |
| varvara_b | 1 |
| yokarine | 4 |
| steven1328 | 1 |
| anjara | 2 |
| roysco | 1 |
| lennonjasonmia@ne... | 2 |
| northwestportland... | 4 |
| bosspad | 10 |
| 31strawberry | 6 |
| nana-tyler | 11 |
+-----+----+
only showing top 20 rows

```

Using SQL

Register a DataFrame as a named temporary table to run SQL.

```

scala> df.registerTempTable("auctions") (1)

scala> val sql = spark.sql("SELECT count(*) AS count FROM auctions")
sql: org.apache.spark.sql.DataFrame = [count: bigint]

```

1. Register a temporary table so SQL queries make sense

You can execute a SQL query on a DataFrame using `sql` operation, but before the query is executed it is optimized by **Catalyst query optimizer**. You can print the physical plan for a DataFrame using the `explain` operation.

```

scala> sql.explain
== Physical Plan ==
TungstenAggregate(key=[], functions=[(count(1),mode=Final,isDistinct=false)], output=[count#148L])
  TungstenExchange SinglePartition
    TungstenAggregate(key=[], functions=[(count(1),mode=Partial,isDistinct=false)], output=[currentCount#156L])
      TungstenProject
        Scan PhysicalRDD[auctionid#49,bid#50,bidtime#51,bidder#52,bidderrate#53,openbid#54,price#55]

scala> sql.show
+----+
|count|
+----+
| 1348|
+----+

scala> val count = sql.collect()(0).getLong(0)
count: Long = 1348

```

Filtering

```

scala> df.show
+---+-----+---+
|name|productId|score|
+---+-----+---+
| aaa|     100| 0.12|
| aaa|     200| 0.29|
| bbb|     200| 0.53|
| bbb|     300| 0.42|
+---+-----+---+

scala> df.filter($"name".like("a%")).show
+---+-----+---+
|name|productId|score|
+---+-----+---+
| aaa|     100| 0.12|
| aaa|     200| 0.29|
+---+-----+---+

```

Handling data in Avro format

Use custom serializer using [spark-avro](#).

Run Spark shell with `--packages com.databricks:spark-avro_2.11:2.0.0` (see [2.0.0 artifact is not in any public maven repo why --repositories is required](#)).

```
./bin/spark-shell --packages com.databricks:spark-avro_2.11:2.0.0 --repositories "http://dl.bintray.com/databricks/maven"
```

And then...

```
val fileRdd = sc.textFile("README.md")
val df = fileRdd.toDF

import org.apache.spark.sql.SaveMode

val outputF = "test.avro"
df.write.mode(SaveMode.Append).format("com.databricks.spark.avro").save(outputF)
```

See [org.apache.spark.sql.SaveMode](#) (and perhaps [org.apache.spark.sql.SaveMode](#) from Scala's perspective).

```
val df = spark.read.format("com.databricks.spark.avro").load("test.avro")
```

Example Datasets

- [eBay online auctions](#)
- [SFPD Crime Incident Reporting system](#)

Row

`Row` is a generic row object with an ordered collection of fields that can be accessed by an [ordinal / an index](#) (aka *generic access by ordinal*), a name (aka *native primitive access*) or using [Scala's pattern matching](#).

Note

`Row` is also called **Catalyst Row**.

`Row` may have an optional [schema](#).

The traits of `Row` :

- `length` or `size` - `Row` knows the number of elements (columns).
- `schema` - `Row` knows the schema

`Row` belongs to `org.apache.spark.sql.Row` package.

```
import org.apache.spark.sql.Row
```

Creating Row — `apply` Factory Method

Caution

FIXME

Field Access by Index — `apply` and `get` methods

Fields of a `Row` instance can be accessed by index (starting from `0`) using `apply` or `get`.

```
scala> val row = Row(1, "hello")
row: org.apache.spark.sql.Row = [1,hello]

scala> row(1)
res0: Any = hello

scala> row.get(1)
res1: Any = hello
```

Note

Generic access by ordinal (using `apply` or `get`) returns a value of type `Any`.

Get Field As Type — `getAs` method

You can query for fields with their proper types using `getAs` with an index

```
val row = Row(1, "hello")

scala> row.getAs[Int](0)
res1: Int = 1

scala> row.getAs[String](1)
res2: String = hello
```

FIXME

Note

`row.getAs[String](null)`

Schema

A `Row` instance can have a schema defined.

Note

Unless you are instantiating `Row` yourself (using [Row Object](#)), a `Row` has always a schema.

Note

It is [RowEncoder](#) to take care of assigning a schema to a `Row` when `toDF` on a [Dataset](#) or when instantiating [DataFrame](#) through [DataFrameReader](#).

Row Object

`Row` companion object offers factory methods to create `Row` instances from a collection of elements (`apply`), a sequence of elements (`fromSeq`) and tuples (`fromTuple`).

```
scala> Row(1, "hello")
res0: org.apache.spark.sql.Row = [1,hello]

scala> Row.fromSeq(Seq(1, "hello"))
res1: org.apache.spark.sql.Row = [1,hello]

scala> Row.fromTuple((0, "hello"))
res2: org.apache.spark.sql.Row = [0,hello]
```

`Row` object can merge `Row` instances.

```
scala> Row.merge(Row(1), Row("hello"))
res3: org.apache.spark.sql.Row = [1,hello]
```

It can also return an empty `Row` instance.

```
scala> Row.empty == Row()
res4: Boolean = true
```

Pattern Matching on Row

`Row` can be used in pattern matching (since [Row Object](#) comes with `unapplySeq`).

```
scala> Row.unapplySeq(Row(1, "hello"))
res5: Some[Seq[Any]] = Some(WrappedArray(1, hello))

Row(1, "hello") match { case Row(key: Int, value: String) =>
  key -> value
}
```

DataSource API — Managing Datasets in External Data Sources

Reading Datasets

Spark SQL can read data from external storage systems like files, Hive tables and JDBC databases through [DataFrameReader](#) interface.

You use [SparkSession](#) to access [DataFrameReader](#) using [read](#) operation.

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder.getOrCreate

val reader = spark.read
```

[DataFrameReader](#) is an interface to create [DataFrames](#) (aka [Dataset\[Row\]](#)) from [files](#), [Hive tables](#) or [tables](#) using [JDBC](#).

```
val people = reader.csv("people.csv")
val cities = reader.format("json").load("cities.json")
```

As of Spark 2.0, [DataFrameReader](#) can read text files using [textFile](#) methods that return [Dataset\[String\]](#) (not [DataFrames](#)).

```
spark.read.textFile("README.md")
```

You can also [define your own custom file formats](#).

```
val countries = reader.format("customFormat").load("countries.cf")
```

There are two operation modes in Spark SQL, i.e. batch and [streaming](#) (part of Spark Structured Streaming).

You can access [DataStreamReader](#) for reading streaming datasets through [SparkSession.readStream](#) method.

```
import org.apache.spark.sql.streaming.DataStreamReader
val stream: DataStreamReader = spark.readStream
```

The available methods in [DataStreamReader](#) are similar to [DataFrameReader](#).

Saving Datasets

Spark SQL can save data to external storage systems like files, Hive tables and JDBC databases through [DataFrameWriter](#) interface.

You use `write` method on a `Dataset` to access `DataFrameWriter`.

```
import org.apache.spark.sql.{DataFrameWriter, Dataset}
val ints: Dataset[Int] = (0 to 5).toDS

val writer: DataFrameWriter[Int] = ints.write
```

`DataFrameWriter` is an interface to persist a [Datasets](#) to an external storage system in a batch fashion.

You can access [DataStreamWriter](#) for writing streaming datasets through `Dataset.writeStream` method.

```
val papers = spark.readStream.text("papers").as[String]

import org.apache.spark.sql.streaming.DataStreamWriter
val writer: DataStreamWriter[String] = papers.writeStream
```

The available methods in `DataStreamWriter` are similar to `DataFrameWriter`.

DataFrameReader — Loading Data From External Data Sources

`DataFrameReader` is a [fluent API](#) to describe the [input data source](#) that will be used to "load" data from an external data source (e.g. [files](#), [tables](#), [JDBC](#) or [Dataset\[String\]](#)).

`DataFrameReader` is [created](#) (available) exclusively using `SparkSession.read`.

```
import org.apache.spark.sql.SparkSession
assert(spark.asInstanceOf[SparkSession])

import org.apache.spark.sql.DataFrameReader
val reader = spark.read
assert(reader.asInstanceOf[DataFrameReader])
```

Table 1. DataFrameReader API

Method	Description
<code>csv</code>	<code>csv(csvDataset: Dataset[String]): DataFrame</code> <code>csv(path: String): DataFrame</code> <code>csv(paths: String*): DataFrame</code>
<code>format</code>	<code>format(source: String): DataFrameReader</code>
<code>jdbc</code>	<code>jdbc(url: String, table: String, predicates: Array[String], connectionProperties: Properties): DataFrame</code> <code>jdbc(url: String, table: String, properties: Properties): DataFrame</code> <code>jdbc(url: String, table: String, columnName: String, lowerBound: Long, upperBound: Long, numPartitions: Int, connectionProperties: Properties): DataFrame</code>
<code>json</code>	<code>json(jsonDataset: Dataset[String]): DataFrame</code> <code>json(path: String): DataFrame</code> <code>json(paths: String*): DataFrame</code>

<code>load</code>	<code>load(): DataFrame</code> <code>load(path: String): DataFrame</code> <code>load(paths: String*): DataFrame</code>
<code>option</code>	<code>option(key: String, value: Boolean): DataFrameReader</code> <code>option(key: String, value: Double): DataFrameReader</code> <code>option(key: String, value: Long): DataFrameReader</code> <code>option(key: String, value: String): DataFrameReader</code>
<code>options</code>	<code>options(options: scala.collection.Map[String, String]): DataFrameReader</code> <code>options(options: java.util.Map[String, String]): DataFrameReader</code>
<code>orc</code>	<code>orc(path: String): DataFrame</code> <code>orc(paths: String*): DataFrame</code>
<code>parquet</code>	<code>parquet(path: String): DataFrame</code> <code>parquet(paths: String*): DataFrame</code>
<code>schema</code>	<code>schema(schemaString: String): DataFrameReader</code> <code>schema(schema: StructType): DataFrameReader</code>
<code>table</code>	<code>table(tableName: String): DataFrame</code>
<code>text</code>	<code>text(path: String): DataFrame</code> <code>text(paths: String*): DataFrame</code>
<code>textFile</code>	<code>textFile(path: String): Dataset[String]</code> <code>textFile(paths: String*): Dataset[String]</code>

`DataFrameReader` supports many [file formats](#) natively and offers the [interface to define custom formats](#).

Note	<code>DataFrameReader</code> assumes parquet data source file format by default that you can change using spark.sql.sources.default configuration property.
-------------	---

After you have described the **loading pipeline** (i.e. the "Extract" part of ETL in Spark SQL), you eventually "trigger" the loading using format-agnostic `load` or format-specific (e.g. `json`, `csv`, `jdbc`) operators.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.DataFrame

// Using format-agnostic load operator
val csvs: DataFrame = spark
  .read
  .format("csv")
  .option("header", true)
  .option("inferSchema", true)
  .load("*.csv")

// Using format-specific load operator
val jsons: DataFrame = spark
  .read
  .json("metrics/*.json")
```

Note

All `methods` of `DataFrameReader` merely describe a process of loading a data and do not trigger a Spark job (until an action is called).

`DataFrameReader` can read text files using `textFile` methods that return typed `Datasets`.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.Dataset
val lines: Dataset[String] = spark
  .read
  .textFile("README.md")
```

Note

Loading datasets using `textFile` methods allows for additional preprocessing before final processing of the string values as `json` or `csv` lines.

(**New in Spark 2.2**) `DataFrameReader` can load datasets from `Dataset[String]` (with lines being complete "files") using format-specific `csv` and `json` operators.

```

val csvLine = "0,Warsaw,Poland"

import org.apache.spark.sql.Dataset
val cities: Dataset[String] = Seq(csvLine).toDS
scala> cities.show
+-----+
|      value|
+-----+
|0,Warsaw,Poland|
+-----+

// Define schema explicitly (as below)
// or
// option("header", true) + option("inferSchema", true)
import org.apache.spark.sql.types.StructType
val schema = new StructType()
  .add($"id".long.copy(nullable = false))
  .add($"city".string)
  .add($"country".string)
scala> schema.printTreeString
root
 |-- id: long (nullable = false)
 |-- city: string (nullable = true)
 |-- country: string (nullable = true)

import org.apache.spark.sql.DataFrame
val citiesDF: DataFrame = spark
  .read
  .schema(schema)
  .csv(cities)
scala> citiesDF.show
+---+---+---+
| id| city|country|
+---+---+---+
| 0|Warsaw| Poland|
+---+---+---+

```

Specifying Format Of Input Data Source— `format` method

```
format(source: String): DataFrameReader
```

You use `format` to configure `DataFrameReader` to use appropriate `source` format.

Supported data formats:

- `json`

- `csv` (since 2.0.0)
- `parquet` (see [Parquet](#))
- `orc`
- `text`
- `jdbc`
- `libsvm` — only when used in `format("libsvm")`

Note	Spark SQL allows for developing custom data source formats .
------	--

Specifying Schema — `schema` method

```
schema(schema: StructType): DataFrameReader
```

`schema` allows for specifying the `schema` of a data source (that the `DataFrameReader` is about to read a dataset from).

```
import org.apache.spark.sql.types.StructType
val schema = new StructType()
  .add($"id".long.copy(nullable = false))
  .add($"city".string)
  .add($"country".string)
scala> schema.printTreeString
root
| -- id: long (nullable = false)
| -- city: string (nullable = true)
| -- country: string (nullable = true)

import org.apache.spark.sql.DataFrameReader
val r: DataFrameReader = spark.read.schema(schema)
```

Note	Some formats can infer schema from datasets (e.g. <code>csv</code> or <code>json</code>) using inferSchema option.
------	---

Tip	Read up on Schema .
-----	-------------------------------------

Specifying Load Options — `option` and `options` Methods

```
option(key: String, value: String): DataFrameReader
option(key: String, value: Boolean): DataFrameReader
option(key: String, value: Long): DataFrameReader
option(key: String, value: Double): DataFrameReader
```

You can also use `options` method to describe different options in a single `Map`.

```
options(options: scala.collection.Map[String, String]): DataFrameReader
```

Loading Datasets from Files (into DataFrames) Using Format-Specific Load Operators

`DataFrameReader` supports the following file formats:

- [JSON](#)
- [CSV](#)
- [parquet](#)
- [ORC](#)
- [text](#)

`json` method

```
json(path: String): DataFrame
json(paths: String*): DataFrame
json(jsonDataset: Dataset[String]): DataFrame
json(jsonRDD: RDD[String]): DataFrame
```

New in **2.0.0**: `prefersDecimal`

`csv` method

```
csv(path: String): DataFrame
csv(paths: String*): DataFrame
csv(csvDataset: Dataset[String]): DataFrame
```

`parquet` method

```
parquet(path: String): DataFrame
parquet(paths: String*): DataFrame
```

The supported options:

- `compression` (default: `snappy`)

New in **2.0.0**: `snappy` is the default Parquet codec. See [\[SPARK-14482\]\[SQL\] Change default Parquet codec from gzip to snappy](#).

The compressions supported:

- `none` or `uncompressed`
- `snappy` - the default codec in Spark **2.0.0**.
- `gzip` - the default codec in Spark before **2.0.0**
- `lzo`

```
val tokens = Seq("hello", "henry", "and", "harry")
  .zipWithIndex
  .map(_.swap)
  .toDF("id", "token")

val parquetWriter = tokens.write
parquetWriter.option("compression", "none").save("hello-none")

// The exception is mostly for my learning purposes
// so I know where and how to find the trace to the compressions
// Sorry...
scala> parquetWriter.option("compression", "unsupported").save("hello-unsupported")
java.lang.IllegalArgumentException: Codec [unsupported] is not available. Available codecs are uncompressed, gzip, lzo, snappy, none.
    at org.apache.spark.sql.execution.datasources.parquet.ParquetOptions.<init>(ParquetOptions.scala:43)
    at org.apache.spark.sql.execution.datasources.parquet.DefaultSource.prepareWrite(ParquetRelation.scala:77)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1$$anonfun$4.apply(InsertIntoHadoopFsRelation.scala:122)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1$$anonfun$4.apply(InsertIntoHadoopFsRelation.scala:122)
    at org.apache.spark.sql.execution.datasources.BaseWriterContainer.driverSideSetup(WriterContainer.scala:103)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1$.apply$mcV$sp(InsertIntoHadoopFsRelation.scala:141)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1$.apply(InsertIntoHadoopFsRelation.scala:116)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1$.apply(InsertIntoHadoopFsRelation.scala:116)
    at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.scala
```

```
a:53)
  at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation.run(InsertI
ntoHadoopFsRelation.scala:116)
  at org.apache.spark.sql.execution.command.ExecutedCommand.sideEffectResult$lzycomput
e(commands.scala:61)
  at org.apache.spark.sql.execution.command.ExecutedCommand.sideEffectResult(commands.s
cala:59)
  at org.apache.spark.sql.execution.command.ExecutedCommand.doExecute(commands.scala:73
)
  at org.apache.spark.sql.execution.SparkPlan$$anonfun$execute$1.apply(SparkPlan.scala:
118)
  at org.apache.spark.sql.execution.SparkPlan$$anonfun$execute$1.apply(SparkPlan.scala:
118)
  at org.apache.spark.sql.execution.SparkPlan$$anonfun$executeQuery$1.apply(SparkPlan.s
cala:137)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:134)
  at org.apache.spark.sql.execution.SparkPlan.execute(SparkPlan.scala:117)
  at org.apache.spark.sql.execution.QueryExecution.toRdd$lzycompute(QueryExecution.sca
la:65)
  at org.apache.spark.sql.execution.QueryExecution.toRdd(QueryExecution.scala:65)
  at org.apache.spark.sql.execution.datasources.DataSource.write(DataSource.scala:390)
  at org.apache.spark.sql.DataFrameWriter.save(DataFrameWriter.scala:247)
  at org.apache.spark.sql.DataFrameWriter.save(DataFrameWriter.scala:230)
  ... 48 elided
```

orc method

```
orc(path: String): DataFrame
orc(paths: String*): DataFrame
```

Optimized Row Columnar (ORC) file format is a highly efficient columnar format to store Hive data with more than 1,000 columns and improve performance. ORC format was introduced in Hive version 0.11 to use and retain the type information from the table definition.

Tip

Read [ORC Files](#) document to learn about the ORC file format.

text method

`text` method loads a text file.

```
text(path: String): DataFrame
text(paths: String*): DataFrame
```

Example

```
val lines: Dataset[String] = spark.read.text("README.md").as[String]

scala> lines.show
+-----+
|      value|
+-----+
|  # Apache Spark|
|          |
|Spark is a fast a...|
|high-level APIs i...|
|supports general ...|
|rich set of highe...|
|Mlib for machine...|
|and Spark Streami...|
|          |
|<http://spark.apa...|
|          |
|          |
|## Online Documen...|
|          |
|You can find the ...|
|guide, on the [pr...|
|and [project wiki...|
|This README file ...|
|          |
|  ## Building Spark|
+-----+
only showing top 20 rows
```

Loading Table to DataFrame — `table` Method

```
table(tableName: String): DataFrame
```

`table` loads the content of the `tableName` table into an untyped [DataFrame](#).

```
scala> spark.catalog.tableExists("t1")
res1: Boolean = true

// t1 exists in the catalog
// let's load it
val t1 = spark.read.table("t1")
```

Note

`table` simply passes the call to [SparkSession.table](#) after making sure that a [user-defined schema](#) has not been specified.

Loading Data From External Table using JDBC Data Source

— jdbc Method

```
jdbc(url: String, table: String, properties: Properties): DataFrame
jdbc(
  url: String,
  table: String,
  predicates: Array[String],
  connectionProperties: Properties): DataFrame
jdbc(
  url: String,
  table: String,
  columnName: String,
  lowerBound: Long,
  upperBound: Long,
  numPartitions: Int,
  connectionProperties: Properties): DataFrame
```

`jdbc` loads data from an external table using the [JDBC data source](#).

Internally, `jdbc` creates a [JDBCOptions](#) from the input `url`, `table` and `extraOptions` with `connectionProperties`.

`jdbc` then creates one `JDBCPartition` per `predicates`.

In the end, `jdbc` requests the [SparkSession](#) to [create a DataFrame](#) for a [JDBCRelation](#) (with `JDBCPartitions` and `JDBCOptions` created earlier).

	<code>jdbc</code> does not support a custom schema and throws an <code>AnalysisException</code> if defined:
Note	User specified schema not supported with `[jdbc]`

	<code>jdbc</code> method uses <code>java.util.Properties</code> (and appears overly Java-centric). Use format("jdbc") instead.
--	--

	Review the exercise Creating DataFrames from Tables using JDBC and PostgreSQL .
--	---

Loading Datasets From Text Files — `textFile` Method

```
textFile(path: String): Dataset[String]
textFile(paths: String*): Dataset[String]
```

`textFile` loads one or many text files into a typed [Dataset\[String\]](#).

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.Dataset
val lines: Dataset[String] = spark
  .read
  .textFile("README.md")
```

Note	<code>textFile</code> are similar to <code>text</code> family of methods in that they both read text files but <code>text</code> methods return untyped <code>DataFrame</code> while <code>textFile</code> return typed <code>Dataset[String]</code> .
------	--

Internally, `textFile` passes calls on to `text` method and [selects](#) the only `value` column before it applies `Encoders.STRING` [encoder](#).

Creating DataFrameReader Instance

`DataFrameReader` takes the following to be created:

- [SparkSession](#)

`DataFrameReader` initializes the [internal properties](#).

Loading Dataset (DataSource API V1) — `loadV1Source` Internal Method

```
loadV1Source(paths: String*): DataFrame
```

`loadV1Source` creates a [DataSource](#) and requests it to [resolve](#) the underlying relation (as a [BaseRelation](#)).

In the end, `loadV1Source` requests [SparkSession](#) to create a `DataFrame` from the [BaseRelation](#).

Note	<code>loadV1Source</code> is used when <code>DataFrameReader</code> is requested to <code>load</code> (and the data source is neither of <code>DataSourceV2</code> type nor a <code>DataSourceReader</code> could not be created).
------	--

"Loading" Data As DataFrame — `load` Method

```
load(): DataFrame
load(path: String): DataFrame
load(paths: String*): DataFrame
```

`load` loads a dataset from a data source (with optional support for multiple `paths`) as an untyped [DataFrame](#).

Internally, `load` [lookupDataSource](#) for the `source`. `load` then branches off per its type (i.e. whether it is of `DataSourceV2` marker type or not).

For a "Data Source V2" data source, `load` ...FIXME

Otherwise, if the `source` is not a "Data Source V2" data source, `load` simply [loadV1Source](#).

`load` throws a `AnalysisException` when the `source format` is `hive`.

Hive data source can only be used with tables, you can not read files of Hive data source directly.

assertNoSpecifiedSchema Internal Method

```
assertNoSpecifiedSchema(operation: String): Unit
```

`assertNoSpecifiedSchema` throws a `AnalysisException` if the `userSpecifiedSchema` is defined.

User specified schema not supported with `operation`

Note	<code>assertNoSpecifiedSchema</code> is used when <code>DataFrameReader</code> is requested to load data using jdbc , table and textFile .
------	--

verifyColumnNameOfCorruptRecord Internal Method

```
verifyColumnNameOfCorruptRecord(
  schema: StructType,
  columnNameOfCorruptRecord: String): Unit
```

`verifyColumnNameOfCorruptRecord` ...FIXME

Note	<code>verifyColumnNameOfCorruptRecord</code> is used when <code>DataFrameReader</code> is requested to load data using json and csv .
------	---

Input Data Source — `source` Internal Property

```
source: String
```

`source` is the name of the input data source (aka *format* or *provider*) that will be used to "load" data (as a DataFrame).

In other words, the [DataFrameReader fluent API](#) is simply to describe the input data source.

The default data source is [parquet](#) per [spark.sql.sources.default](#) configuration property.

`source` is usually specified using [format](#) method.

Important	<p><code>source</code> must not be <code>hive</code> or an <code>AnalysisException</code> is thrown:</p> <p>Hive data source can only be used with tables, you can not read files of H</p>
-----------	--

Once defined explicitly (using [format](#) method) or implicitly ([spark.sql.sources.default](#) configuration property), `source` is resolved using [DataSource](#) utility.

Note	<p><code>source</code> is used exclusively when <code>DataFrameReader</code> is requested to "load" data (as a DataFrame) (explicitly or using loadV1Source).</p>
------	---

Internal Properties

Name	Description
<code>extraOptions</code>	Used when...FIXME
<code>userSpecifiedSchema</code>	<p>Optional used-specified schema (default: <code>None</code>, i.e. <code>undefined</code>)</p> <p>Set when <code>DataFrameReader</code> is requested to set a schema, load a data from an external data source, loadV1Source (when creating a DataSource), and load a data using json and csv file formats</p> <p>Used when <code>DataFrameReader</code> is requested to assertNoSpecifiedSchema (while loading data using jdbc, table and textFile)</p>

DataFrameWriter — Saving Data To External Data Sources

`DataFrameWriter` is the [interface](#) to describe how data (as the result of executing a structured query) should be [saved to an external data source](#).

Table 1. DataFrameWriter API / Writing Operators

Method	Description
<code>bucketBy</code>	<code>bucketBy(numBuckets: Int, colName: String, colNames: String*): DataFrameWriter[T]</code>
<code>csv</code>	<code>csv(path: String): Unit</code>
<code>format</code>	<code>format(source: String): DataFrameWriter[T]</code>
<code>insertInto</code>	<code>insertInto(tableName: String): Unit</code> Inserts (the results of) a structured query (<code>DataFrame</code>) into a table
<code>jdbc</code>	<code>jdbc(url: String, table: String, connectionProperties: Properties): Unit</code>
<code>json</code>	<code>json(path: String): Unit</code>
<code>mode</code>	<code>mode(saveMode: SaveMode): DataFrameWriter[T]</code> <code>mode(saveMode: String): DataFrameWriter[T]</code>
<code>option</code>	<code>option(key: String, value: String): DataFrameWriter[T]</code> <code>option(key: String, value: Boolean): DataFrameWriter[T]</code> <code>option(key: String, value: Long): DataFrameWriter[T]</code> <code>option(key: String, value: Double): DataFrameWriter[T]</code>
<code>options</code>	<code>options(options: scala.collection.Map[String, String]): DataFrameWriter[T]</code>

<code>orc</code>	<code>orc(path: String): Unit</code>
<code>parquet</code>	<code>parquet(path: String): Unit</code>
<code>partitionBy</code>	<code>partitionBy(colNames: String*): DataFrameWriter[T]</code>
<code>save</code>	<code>save(): Unit</code> <code>save(path: String): Unit</code> Saves a <code>DataFrame</code> (i.e. writes the result of executing a structured query) to a <code>source</code>
<code>saveAsTable</code>	<code>saveAsTable(tableName: String): Unit</code>
<code>sortBy</code>	<code>sortBy(colName: String, colNames: String*): DataFrameWriter[T]</code>
<code>text</code>	<code>text(path: String): Unit</code>

`DataFrameWriter` is available using `Dataset.write` operator.

```
scala> :type df
org.apache.spark.sql.DataFrame

val writer = df.write

scala> :type writer
org.apache.spark.sql.DataFrameWriter[org.apache.spark.sql.Row]
```

`DataFrameWriter` supports many [file formats](#) and [JDBC databases](#). It also allows for plugging in [new formats](#).

`DataFrameWriter` defaults to [parquet](#) data source format. You can change the default format using `spark.sql.sources.default` configuration property or `format` or the format-specific methods.

```
// see above for writer definition

// Save dataset in Parquet format
writer.save(path = "nums")

// Save dataset in JSON format
writer.format("json").save(path = "nums-json")

// Alternatively, use format-specific method
write.json(path = "nums-json")
```

In the end, you trigger the actual saving of the content of a `dataset` (i.e. the result of executing a structured query) using `save` method.

```
writer.save
```

`DataFrameWriter` uses [internal mutable attributes](#) to build a properly-defined "*write specification*" for `insertInto`, `save` and `saveAsTable` methods.

Table 2. Internal Attributes and Corresponding Setters

Attribute	Setters
<code>source</code>	<code>format</code>
<code>mode</code>	<code>mode</code>
<code>extraOptions</code>	<code>option</code> , <code>options</code> , <code>save</code>
<code>partitioningColumns</code>	<code>partitionBy</code>
<code>bucketColumnNames</code>	<code>bucketBy</code>
<code>numBuckets</code>	<code>bucketBy</code>
<code>sortColumnNames</code>	<code>sortBy</code>

Note	<code>DataFrameWriter</code> is a type constructor in Scala that keeps an internal reference to the source <code>DataFrame</code> for the whole lifecycle (starting right from the moment it was created).
------	--

Note	Spark Structured Streaming's <code>DataStreamWriter</code> is responsible for writing the content of streaming Datasets in a streaming fashion.
------	---

Executing Logical Command(s) — `runCommand` Internal Method

```
runCommand(session: SparkSession, name: String)(command: LogicalPlan): Unit
```

`runCommand` uses the input `SparkSession` to access the `SessionState` that is in turn requested to [execute the logical command](#) (that simply creates a `QueryExecution`).

`runCommand` records the current time (start time) and uses the `SQLExecution` helper object to [execute the action \(under a new execution id\)](#) that simply requests the `QueryExecution` for the `RDD[InternalRow]` (and triggers execution of logical commands).

Tip

Use web UI's SQL tab to see the execution or a `SparkListener` to be notified when the execution is started and finished. The `SparkListener` should intercept `SparkListenerSQLExecutionStart` and `SparkListenerSQLExecutionEnd` events.

`runCommand` records the current time (end time).

In the end, `runCommand` uses the input `SparkSession` to access the `ExecutionListenerManager` and requests it to `onSuccess` (with the input `name`, the `QueryExecution` and the duration).

In case of any exceptions, `runCommand` requests the `ExecutionListenerManager` to `onFailure` (with the exception) and (re)throws it.

Note

`runCommand` is used when `DataStreamWriter` is requested to [save the rows of a structured query \(a DataFrame\) to a data source](#) (and indirectly [executing a logical command for writing to a data source V1](#)), [insert the rows of a structured streaming \(a DataFrame\) into a table](#) and [create a table](#) (that is used exclusively for `saveAsTable`).

Saving Rows of Structured Streaming (DataFrame) to Table — `saveAsTable` Method

```
saveAsTable(tableName: String): Unit
// PRIVATE API
saveAsTable(tableIdent: TableIdentifier): Unit
```

`saveAsTable` saves the content of a `DataFrame` to the `tableName` table.

```

val ids = spark.range(5)
ids.write.
  option("path", "/tmp/five_ids").
  saveAsTable("five_ids")

// Check out if saveAsTable as five_ids was successful
val q = spark.catalog.listTables.filter($"name" === "five_ids")
scala> q.show
+-----+-----+-----+-----+
|    name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|five_ids| default|      null| EXTERNAL|     false|
+-----+-----+-----+-----+

```

Internally, `saveAsTable` requests the current `ParserInterface` to [parse the input table name](#).

Note	<code>saveAsTable</code> uses the internal DataFrame to access the SparkSession that is used to access the SessionState and in the end the ParserInterface .
------	--

`saveAsTable` then requests the `SessionCatalog` to [check whether the table exists or not](#).

Note	<code>saveAsTable</code> uses the internal DataFrame to access the SparkSession that is used to access the SessionState and in the end the SessionCatalog .
------	---

In the end, `saveAsTable` branches off per whether the table exists or not and the [save mode](#).

Table 3. `saveAsTable`'s Behaviour per Save Mode

Does table exist?	Save Mode	Behaviour
yes	Ignore	Does nothing
yes	ErrorIfExists	Reports an <code>AnalysisException</code> with <code>Table [tableIdent]</code> already exists. error message
yes	Overwrite	FIXME
anything	anything	<code>createTable</code>

Saving Rows of Structured Query (DataFrame) to Data Source — `save` Method

```
save(): Unit
```

`save` saves the rows of a structured query (a [Dataset](#)) to a data source.

Internally, `save` uses `DataSource` to look up the class of the requested data source (for the `source` option and the [SQLConf](#)).

`save` uses [SparkSession](#) to access the [SessionState](#) that is in turn used to access the [SQLConf](#).

Note

```
val df: DataFrame = ???  
df.sparkSession.sessionState.conf
```

If the class is a [DataSourceV2](#)...FIXME

Otherwise, if not a [DataSourceV2](#), `save` simply [saveToV1Source](#).

`save` does not support saving to Hive (i.e. the `source` is `hive`) and throws an `AnalysisException` when requested so.

Hive data source can only be used with tables, you can not write files of Hive data source directly.

`save` does not support bucketing (i.e. when the `numBuckets` or `sortColumnNames` options are defined) and throws an `AnalysisException` when requested so.

```
'[operation]' does not support bucketing right now
```

Saving Data to Table Using JDBC Data Source — `jdbc` Method

```
jdbc(url: String, table: String, connectionProperties: Properties): Unit
```

`jdbc` method saves the content of the `DataFrame` to an external database table via JDBC.

You can use `mode` to control **save mode**, i.e. what happens when an external table exists when `save` is executed.

It is assumed that the `jdbc` save pipeline is not [partitioned](#) and [bucketed](#).

All `options` are overriden by the input `connectionProperties`.

The required options are:

- `driver` which is the class name of the JDBC driver (that is passed to Spark's own `DriverRegistry.register` and later used to `connect(url, properties)`).

When `table` exists and the `override save mode` is in use, `DROP TABLE table` is executed.

It creates the input `table` (using `CREATE TABLE table (schema)` where `schema` is the schema of the `DataFrame`).

bucketBy Method

```
bucketBy(numBuckets: Int, colName: String, colNames: String*): DataFrameWriter[T]
```

`bucketBy` simply sets the internal `numBuckets` and `bucketColumnNames` to the input `numBuckets` and `colName` with `colNames`, respectively.

```
val df = spark.range(5)
import org.apache.spark.sql.DataFrameWriter
val writer: DataFrameWriter[java.lang.Long] = df.write

val bucketedTable = writer.bucketBy(numBuckets = 8, "col1", "col2")

scala> :type bucketedTable
org.apache.spark.sql.DataFrameWriter[Long]
```

partitionBy Method

```
partitionBy(colNames: String*): DataFrameWriter[T]
```

Caution	FIXME
---------	-------

Specifying Save Mode — mode Method

```
mode(saveMode: String): DataFrameWriter[T]
mode(saveMode: SaveMode): DataFrameWriter[T]
```

`mode` defines the behaviour of `save` when an external file or table (Spark writes to) already exists, i.e. `SaveMode`.

Table 4. Types of SaveMode

Name	Description
Append	Records are appended to existing data.
ErrorIfExists	Exception is thrown.
Ignore	Do not save the records and not change the existing data in any way.
Overwrite	Existing data is overwritten by new records.

Specifying Sorting Columns — `sortBy` Method

```
sortBy(colName: String, colNames: String*): DataFrameWriter[T]
```

`sortBy` simply sets `sorting columns` to the input `colName` and `colNames` column names.

Note	<code>sortBy</code> must be used together with <code>bucketBy</code> or <code>DataFrameWriter</code> reports an <code>IllegalArgumentException</code> .
------	---

Note	<code>assertNotBucketed</code> asserts that bucketing is not used by some methods.
------	--

Specifying Writer Configuration — `option` Method

```
option(key: String, value: Boolean): DataFrameWriter[T]
option(key: String, value: Double): DataFrameWriter[T]
option(key: String, value: Long): DataFrameWriter[T]
option(key: String, value: String): DataFrameWriter[T]
```

`option` ...FIXME

Specifying Writer Configuration — `options` Method

```
options(options: scala.collection.Map[String, String]): DataFrameWriter[T]
```

`options` ...FIXME

Writing DataFrames to Files

Caution	FIXME
---------	-------

Specifying Data Source (by Alias or Fully-Qualified Class Name) — `format` Method

```
format(source: String): DataFrameWriter[T]
```

`format` simply sets the `source` internal property.

Parquet

Caution	FIXME
---------	-------

Note	Parquet is the default data source format.
------	--

Inserting Rows of Structured Streaming (DataFrame) into Table — `insertInto` Method

```
insertInto(tableName: String): Unit (1)
insertInto(tableIdent: TableIdentifier): Unit
```

1. Parses `tableName` and calls the other `insertInto` with a `TableIdentifier`

`insertInto` inserts the content of the `DataFrame` to the specified `tableName` table.

Note	<code>insertInto</code> ignores column names and just uses a position-based resolution, i.e. the order (not the names!) of the columns in (the output of) the Dataset matters.
------	--

Internally, `insertInto` creates an `InsertIntoTable` logical operator (with `UnresolvedRelation` operator as the only child) and `executes` it right away (that submits a Spark job).

Details for Query 12

Submitted Time: 2018/01/11 11:26:19

Duration: 0.4 s

Succeeded Jobs: 8

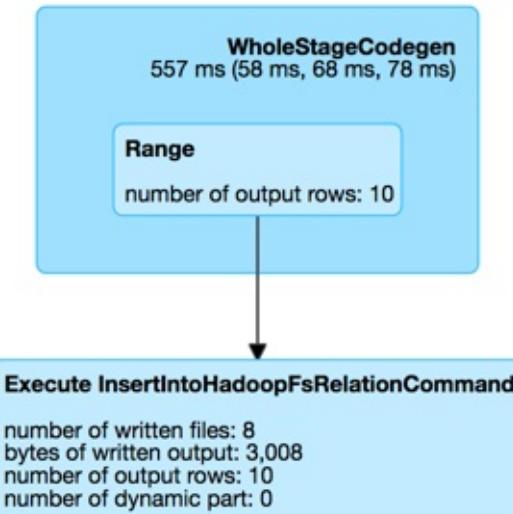


Figure 1. DataFrameWrite.insertInto Executes SQL Command (as a Spark job)

`insertInto` reports a `AnalysisException` for bucketed DataFrames, i.e. `buckets` or `sortColumnNames` are defined.

'`insertInto`' does not support bucketing right now

```

val writeSpec = spark.range(4).
  write.
  bucketBy(numBuckets = 3, colName = "id")
scala> writeSpec.insertInto("t1")
org.apache.spark.sql.AnalysisException: 'insertInto' does not support bucketing right
now;
  at org.apache.spark.sql.DataFrameWriter.assertNotBucketed(DataFrameWriter.scala:334)
  at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:302)
  at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:298)
  ... 49 elided
  
```

`insertInto` reports a `AnalysisException` for partitioned DataFrames, i.e. `partitioningColumns` is defined.

`insertInto()` can't be used together with `partitionBy()`. Partition columns have already been defined for the table. It is not necessary to use `partitionBy()`.

```

val writeSpec = spark.range(4).
  write.
  partitionBy("id")
scala> writeSpec.insertInto("t1")
org.apache.spark.sql.AnalysisException: insertInto() can't be used together with partitionBy(). Partition columns have already be defined for the table. It is not necessary to use partitionBy();
  at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:305)
  at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:298)
... 49 elided

```

getBucketSpec Internal Method

`getBucketSpec: Option[BucketSpec]`

`getBucketSpec` returns a new `BucketSpec` if `numBuckets` was defined (with `bucketColumnNames` and `sortColumnNames`).

`getBucketSpec` throws an `IllegalArgumentException` when `numBuckets` are not defined when `sortColumnNames` are.

`sortBy` must be used together with `bucketBy`

Note	<code>getBucketSpec</code> is used exclusively when <code>DataFrameWriter</code> is requested to create a table .
------	---

Creating Table — `createTable` Internal Method

`createTable(tableIdent: TableIdentifier): Unit`

`createTable` builds a `CatalogStorageFormat` per `extraOptions`.

`createTable` assumes `CatalogTableType.EXTERNAL` when `location URI` of `CatalogStorageFormat` is defined and `CatalogTableType.MANAGED` otherwise.

`createTable` creates a `CatalogTable` (with the `bucketSpec` per `getBucketSpec`).

In the end, `createTable` creates a `CreateTable` logical command (with the `CatalogTable`, `mode` and the `logical query plan` of the `dataset`) and `runs` it.

Note	<code>createTable</code> is used when <code>DataFrameWriter</code> is requested to saveAsTable .
------	--

assertNotBucketed Internal Method

```
assertNotBucketed(operation: String): Unit
```

`assertNotBucketed` simply throws an `AnalysisException` if either `numBuckets` or `sortColumnNames` internal property is defined:

```
'[operation]' does not support bucketing right now
```

Note

`assertNotBucketed` is used when `DataFrameWriter` is requested to `save`, `insertInto` and `jdbc`.

Executing Logical Command for Writing to Data Source V1 — saveToV1Source Internal Method

```
saveToV1Source(): Unit
```

`saveToV1Source` creates a `DataSource` (for the `source` class name, the `partitioningColumns` and the `extraOptions`) and requests it for the `logical command for writing` (with the `mode` and the `analyzed logical plan` of the structured query).

Note

While requesting the `analyzed logical plan` of the structured query, `saveToV1Source` triggers execution of logical commands.

In the end, `saveToV1Source` runs the `logical command for writing`.

Note

The `logical command for writing` can be one of the following:

- A `SaveIntoDataSourceCommand` for `CreatableRelationProviders`
- An `InsertIntoHadoopFsRelationCommand` for `FileFormats`

Note

`saveToV1Source` is used exclusively when `DataFrameWriter` is requested to save the rows of a structured query (a `DataFrame`) to a data source (for all but `DataSourceV2` writers with `WriteSupport`).

assertNotPartitioned Internal Method

```
assertNotPartitioned(operation: String): Unit
```

`assertNotPartitioned` ...FIXME

Note`assertNotPartitioned` is used when...FIXME**csv Method**`csv(path: String): Unit``csv` ...FIXME**json Method**`json(path: String): Unit``json` ...FIXME**orc Method**`orc(path: String): Unit``orc` ...FIXME**parquet Method**`parquet(path: String): Unit``parquet` ...FIXME**text Method**`text(path: String): Unit``text` ...FIXME**partitionBy Method**`partitionBy(colNames: String*): DataFrameWriter[T]`

`partitionBy` simply sets the `partitioningColumns` internal property.

Dataset API — Dataset Operators

Dataset API is a [set of operators](#) with typed and untyped transformations, and actions to work with a structured query (as a [Dataset](#)) as a whole.

Table 1. Dataset Operators (Transformations and Action)

Operator	Description
agg	<pre>agg(aggExpr: (String, String), aggExprs: (String, agg(expr: Column, exprs: Column*): DataFrame agg(exprs: Map[String, String]): DataFrame</pre> <p>An untyped transformation</p>
alias	<pre>alias(alias: String): Dataset[T] alias(alias: Symbol): Dataset[T]</pre> <p>A typed transformation that is a mere synonym of as.</p>
apply	<pre>apply(colName: String): Column</pre> <p>An untyped transformation to select a column based <code>Dataset</code> onto a <code>Column</code>)</p>
as	<pre>as(alias: String): Dataset[T] as(alias: Symbol): Dataset[T]</pre> <p>A typed transformation</p>
as	<pre>as[U : Encoder]: Dataset[U]</pre> <p>A typed transformation to enforce a type, i.e. marking given data type (<i>data type conversion</i>). <code>as</code> simply cl passed into typed operations (e.g. map) and does no that are not present in the specified class.</p>
cache	<pre>cache(): this.type</pre> <p>A basic action that is a mere synonym of persist.</p>

<code>checkpoint</code>	<pre>checkpoint(): Dataset[T] checkpoint(eager: Boolean): Dataset[T]</pre>
	A basic action to checkpoint the <code>Dataset</code> in a reliable compliant file system, e.g. Hadoop HDFS or Amazon S3.
<code>coalesce</code>	<pre>coalesce(numPartitions: Int): Dataset[T]</pre>
	A typed transformation to repartition a Dataset
<code>col</code>	<pre>col(colName: String): Column</pre>
	An untyped transformation to create a column (referencing a column by name)
<code>collect</code>	<pre>collect(): Array[T]</pre>
	An action
<code>colRegex</code>	<pre>colRegex(colName: String): Column</pre>
	An untyped transformation to create a column (referencing a column by name specified as a regex)
<code>columns</code>	<pre>columns: Array[String]</pre>
	A basic action
<code>count</code>	<pre>count(): Long</pre>
	An action to count the number of rows
<code>createGlobalTempView</code>	<pre>createGlobalTempView(viewName: String): Unit</pre>
	A basic action
<code>createOrReplaceGlobalTempView</code>	<pre>createOrReplaceGlobalTempView(viewName: String): Unit</pre>
	A basic action

<code>createOrReplaceTempView</code>	<pre>createOrReplaceTempView(viewName: String): Unit</pre> <p>A basic action</p>
<code>createTempView</code>	<pre>createTempView(viewName: String): Unit</pre> <p>A basic action</p>
<code>crossJoin</code>	<pre>crossJoin(right: Dataset[_]): DataFrame</pre> <p>An untyped transformation</p>
<code>cube</code>	<pre>cube(cols: Column*): RelationalGroupedDataset cube(col1: String, cols: String*): RelationalGroupedDataset</pre> <p>An untyped transformation</p>
<code>describe</code>	<pre>describe(cols: String*): DataFrame</pre> <p>An action</p>
<code>distinct</code>	<pre>distinct(): Dataset[T]</pre> <p>A typed transformation that is a mere synonym of <code>dropDuplicates</code> (the <code>dataset</code>)</p>
<code>drop</code>	<pre>drop(colName: String): DataFrame drop(colNames: String*): DataFrame drop(col: Column): DataFrame</pre> <p>An untyped transformation</p>
<code>dropDuplicates</code>	<pre>dropDuplicates(): Dataset[T] dropDuplicates(colNames: Array[String]): Dataset[T] dropDuplicates(colNames: Seq[String]): Dataset[T] dropDuplicates(col1: String, cols: String*): Dataset[T]</pre> <p>A typed transformation</p>
<code>dtypes</code>	<pre>dtypes: Array[(String, String)]</pre>

	A basic action
except	<code>except(other: Dataset[T]): Dataset[T]</code>
	A typed transformation
exceptAll	<code>exceptAll(other: Dataset[T]): Dataset[T]</code>
	(New in 2.4.4) A typed transformation
explain	<code>explain(): Unit</code> <code>explain(extended: Boolean): Unit</code>
	A basic action to display the logical and physical plan logical and physical plans (with optional cost and cod output
filter	<code>filter(condition: Column): Dataset[T]</code> <code>filter(conditionExpr: String): Dataset[T]</code> <code>filter(func: T => Boolean): Dataset[T]</code>
	A typed transformation
first	<code>first(): T</code>
	An action that is a mere synonym of <code>head</code>
flatMap	<code>flatMap[U : Encoder](func: T => TraversableOnce[U])</code>
	A typed transformation
foreach	<code>foreach(f: T => Unit): Unit</code>
	An action
foreachPartition	<code>foreachPartition(f: Iterator[T] => Unit): Unit</code>
	An action

groupBy	<pre>groupBy(cols: Column*): RelationalGroupedDataset groupBy(col1: String, cols: String*): RelationalGr</pre>
	An untyped transformation
groupByKey	<pre>groupByKey[K: Encoder](func: T => K): KeyValueGrou</pre>
	A typed transformation
head	<pre>head(): T (1) head(n: Int): Array[T]</pre>
	1. Uses <code>1</code> for <code>n</code>
	An action
hint	<pre>hint(name: String, parameters: Any*): Dataset[T]</pre>
	A basic action to specify a hint (and optional paramet
inputFiles	<pre>inputFiles: Array[String]</pre>
	A basic action
intersect	<pre>intersect(other: Dataset[T]): Dataset[T]</pre>
	A typed transformation
intersectAll	<pre>intersectAll(other: Dataset[T]): Dataset[T]</pre>
	(New in 2.4.4) A typed transformation
isEmpty	<pre>isEmpty: Boolean</pre>
	(New in 2.4.4) A basic action
isLocal	<pre>isLocal: Boolean</pre>
	A basic action

isStreaming	<code>isStreaming: Boolean</code>
join	<code>join(right: Dataset[_]): DataFrame</code> <code>join(right: Dataset[_], usingColumn: String): Dataset[Type]</code> <code>join(right: Dataset[_], usingColumns: Seq[String]): Dataset[Type]</code> <code>join(right: Dataset[_], usingColumns: Seq[String], joinType: String): Dataset[Type]</code> <code>join(right: Dataset[_], joinExprs: Column): Dataset[Type]</code> <code>join(right: Dataset[_], joinExprs: Column, joinType: String): Dataset[Type]</code>
	An untyped transformation
joinWith	<code>joinWith[U](other: Dataset[U], condition: Column): Dataset[Type]</code> <code>joinWith[U](other: Dataset[U], condition: Column, joinType: String): Dataset[Type]</code>
	A typed transformation
limit	<code>limit(n: Int): Dataset[T]</code>
	A typed transformation
localCheckpoint	<code>localCheckpoint(): Dataset[T]</code> <code>localCheckpoint(eager: Boolean): Dataset[T]</code>
	A basic action to checkpoint the <code>Dataset</code> locally on each partition.
map	<code>map[U: Encoder](func: T => U): Dataset[U]</code>
	A typed transformation
mapPartitions	<code>mapPartitions[U : Encoder](func: Iterator[T] => Iterator[U]): Dataset[U]</code>
	A typed transformation
na	<code>na: DataFrameNaFunctions</code>
	An untyped transformation
orderBy	<code>orderBy(sortExprs: Column*): Dataset[T]</code> <code>orderBy(sortCol: String, sortCols: String*): Dataset[T]</code>
	A typed transformation

	<pre><code>persist(): this.type persist(newLevel: StorageLevel): this.type</code></pre>
persist	<p>A basic action to persist the <code>Dataset</code></p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>Note</p> <p>Although its category <code>persist</code> is not an <code>Action</code>, it means executing <i>anything</i> in a Spark cluster (on executors). It acts only as a marker to prevent an action is really executed.</p> </div>
printSchema	<pre><code>printSchema(): Unit</code></pre> <p>A basic action</p>
randomSplit	<pre><code>randomSplit(weights: Array[Double]): Array[Dataset[T]] randomSplit(weights: Array[Double], seed: Long): Array[Dataset[T]]</code></pre> <p>A typed transformation to split a <code>Dataset</code> randomly into multiple datasets.</p>
rdd	<pre><code>rdd: RDD[T]</code></pre> <p>A basic action</p>
reduce	<pre><code>reduce(func: (T, T) => T): T</code></pre> <p>An action to reduce the records of the <code>Dataset</code> using the specified function.</p>
repartition	<pre><code>repartition(partitionExprs: Column*): Dataset[T] repartition(numPartitions: Int): Dataset[T] repartition(numPartitions: Int, partitionExprs: Column*): Dataset[T]</code></pre> <p>A typed transformation to repartition a <code>Dataset</code>.</p>
repartitionByRange	<pre><code>repartitionByRange(partitionExprs: Column*): Dataset[T] repartitionByRange(numPartitions: Int, partitionExprs: Column*): Dataset[T]</code></pre> <p>A typed transformation</p>
rollup	<pre><code>rollup(cols: Column*): RelationalGroupedDataset rollup(col1: String, cols: String*): RelationalGroupedDataset</code></pre> <p>An untyped transformation</p>

sample	<pre>sample(withReplacement: Boolean, fraction: Double) sample(withReplacement: Boolean, fraction: Double, sample(fraction: Double): Dataset[T] sample(fraction: Double, seed: Long): Dataset[T]</pre>
	A typed transformation
schema	<pre>schema: StructType</pre>
	A basic action
select	<pre>select(cols: Column*): DataFrame select(col: String, cols: String*): DataFrame select[U1](c1: TypedColumn[T, U1]): Dataset[U1] select[U1, U2](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2]): Dataset[(U1, U2)] select[U1, U2, U3](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2], c3: TypedColumn[T, U3]): Dataset[(U1, U2, U3)] select[U1, U2, U3, U4](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2], c3: TypedColumn[T, U3], c4: TypedColumn[T, U4]): Dataset[(U1, U2, U3, U4)] select[U1, U2, U3, U4, U5](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2], c3: TypedColumn[T, U3], c4: TypedColumn[T, U4], c5: TypedColumn[T, U5]): Dataset[(U1, U2, U3, U4, U5)]</pre>
	An (untyped and typed) transformation
selectExpr	<pre>selectExpr(exprs: String*): DataFrame</pre>
	An untyped transformation
show	<pre>show(): Unit show(truncate: Boolean): Unit show(numRows: Int): Unit show(numRows: Int, truncate: Boolean): Unit show(numRows: Int, truncate: Int): Unit show(numRows: Int, truncate: Int, vertical: Boolean): Unit</pre>
	An action
sort	<pre>sort(sortExprs: Column*): Dataset[T] sort(sortCol: String, sortCols: String*): Dataset[T]</pre>

	A typed transformation to sort elements globally (across partitions) or locally within partitions. This transformation is part of the <code>Dataset</code> API.
<code>sortWithinPartitions</code>	<pre>sortWithinPartitions(sortExprs: Column*): Dataset[Row] sortWithinPartitions(sortCol: String, sortCols: Seq[String]): Dataset[Row]</pre> <p>A typed transformation to sort elements within partitions. This transformation is part of the <code>Dataset</code> API.</p>
<code>stat</code>	<pre>stat: DataFrameStatFunctions</pre> <p>An untyped transformation</p>
<code>storageLevel</code>	<pre>storageLevel: StorageLevel</pre> <p>A basic action</p>
<code>summary</code>	<pre>summary(statistics: String*): DataFrame</pre> <p>An action to calculate statistics (e.g. <code>count</code> , <code>mean</code> , <code>50%</code> , <code>75%</code> <code>percentiles</code>)</p>
<code>take</code>	<pre>take(n: Int): Array[T]</pre> <p>An action to take the first records of a Dataset</p>
<code>toDF</code>	<pre>toDF(): DataFrame toDF(colNames: String*): DataFrame</pre> <p>A basic action to convert a Dataset to a DataFrame</p>
<code>toJSON</code>	<pre>toJSON: Dataset[String]</pre> <p>A typed transformation</p>
<code>toLocalIterator</code>	<pre>toLocalIterator(): java.util.Iterator[T]</pre> <p>An action that returns an iterator with all rows in the Dataset. This action uses as much memory as the largest partition in the Dataset.</p>

transform	<pre>transform[U](t: Dataset[T] => Dataset[U]): Dataset[U]</pre>
	A typed transformation for chaining custom transform
union	<pre>union(other: Dataset[T]): Dataset[T]</pre>
	A typed transformation
unionByName	<pre>unionByName(other: Dataset[T]): Dataset[T]</pre>
	A typed transformation
unpersist	<pre>unpersist(): this.type (1) unpersist(blocking: Boolean): this.type</pre>
	1. Uses <code>unpersist</code> with <code>blocking</code> disabled (<code>false</code>)
	A basic action to unpersist the <code>Dataset</code>
where	<pre>where(condition: Column): Dataset[T] where(conditionExpr: String): Dataset[T]</pre>
	A typed transformation
withColumn	<pre>withColumn(colName: String, col: Column): DataFrame</pre>
	An untyped transformation
withColumnRenamed	<pre>withColumnRenamed(existingName: String, newName: String): DataFrame</pre>
	An untyped transformation
write	<pre>write: DataFrameWriter[T]</pre>
	A basic action that returns a <code>DataFrameWriter</code> for saving (or streaming) <code>dataset</code> out to an external storage

Dataset API — Typed Transformations

Typed transformations are part of the Dataset API for transforming a `Dataset` with an [Encoder](#) (except the [RowEncoder](#)).

Note

Typed transformations are the methods in the `Dataset` Scala class that are grouped in `typedrel` group name, i.e. `@group typedrel`.

Table 1. Dataset API's Typed Transformations

Transformation	Description
<code>alias</code>	<code>alias(alias: String): Dataset[T]</code> <code>alias(alias: Symbol): Dataset[T]</code>
<code>as</code>	<code>as(alias: String): Dataset[T]</code> <code>as(alias: Symbol): Dataset[T]</code>
<code>as</code>	<code>as[U : Encoder]: Dataset[U]</code>
<code>coalesce</code>	Repartitions a Dataset <code>coalesce(numPartitions: Int): Dataset[T]</code>
<code>distinct</code>	<code>distinct(): Dataset[T]</code>
<code>dropDuplicates</code>	<code>dropDuplicates(): Dataset[T]</code> <code>dropDuplicates(colNames: Array[String]): Dataset[T]</code> <code>dropDuplicates(colNames: Seq[String]): Dataset[T]</code> <code>dropDuplicates(col1: String, cols: String*): Dataset[T]</code>
<code>except</code>	<code>except(other: Dataset[T]): Dataset[T]</code>
<code>filter</code>	<code>filter(condition: Column): Dataset[T]</code> <code>filter(conditionExpr: String): Dataset[T]</code> <code>filter(func: T => Boolean): Dataset[T]</code>

flatMap	<code>flatMap[U : Encoder](func: T => TraversableOnce[U]): Dataset[U]</code>
groupByKey	<code>groupByKey[K: Encoder](func: T => K): KeyValueGroupedDataset[K,</code>
intersect	<code>intersect(other: Dataset[T]): Dataset[T]</code>
joinWith	<code>joinWith[U](other: Dataset[U], condition: Column): Dataset[(T, U)] joinWith[U](other: Dataset[U], condition: Column, joinType: String): Dataset[(T, U)]</code>
limit	<code>limit(n: Int): Dataset[T]</code>
map	<code>map[U: Encoder](func: T => U): Dataset[U]</code>
mapPartitions	<code>mapPartitions[U : Encoder](func: Iterator[T] => Iterator[U]): Dataset[U]</code>
orderBy	<code>orderBy(sortExprs: Column*): Dataset[T] orderBy(sortCol: String, sortCols: String*): Dataset[T]</code>
randomSplit	<code>randomSplit(weights: Array[Double]): Array[Dataset[T]] randomSplit(weights: Array[Double], seed: Long): Array[Dataset[T]]</code>
repartition	<code>repartition(partitionExprs: Column*): Dataset[T] repartition(numPartitions: Int): Dataset[T] repartition(numPartitions: Int, partitionExprs: Column*): Dataset[T]</code>
repartitionByRange	<code>repartitionByRange(partitionExprs: Column*): Dataset[T] repartitionByRange(numPartitions: Int, partitionExprs: Column*): Dataset[T]</code>
sample	<code>sample(withReplacement: Boolean, fraction: Double): Dataset[T] sample(withReplacement: Boolean, fraction: Double, seed: Long): Dataset[T] sample(fraction: Double): Dataset[T] sample(fraction: Double, seed: Long): Dataset[T]</code>

<code>select</code>	<pre> select[U1](c1: TypedColumn[T, U1]): Dataset[U1] select[U1, U2](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2]): Dataset[U1, U2] select[U1, U2, U3](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2], c3: TypedColumn[T, U3]): Dataset[(U1, U2, U3)] select[U1, U2, U3, U4](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2], c3: TypedColumn[T, U3], c4: TypedColumn[T, U4]): Dataset[(U1, U2, U3, U4)] select[U1, U2, U3, U4, U5](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2], c3: TypedColumn[T, U3], c4: TypedColumn[T, U4], c5: TypedColumn[T, U5]): Dataset[(U1, U2, U3, U4, U5)] </pre>
<code>sort</code>	<pre> sort(sortExprs: Column*): Dataset[T] sort(sortCol: String, sortCols: String*): Dataset[T] </pre>
<code>sortWithinPartitions</code>	<pre> sortWithinPartitions(sortExprs: Column*): Dataset[T] sortWithinPartitions(sortCol: String, sortCols: String*): Dataset[T] </pre>
<code>toJSON</code>	<code>toJSON: Dataset[String]</code>
<code>transform</code>	<code>transform[U](t: Dataset[T] => Dataset[U]): Dataset[U]</code>
<code>union</code>	<code>union(other: Dataset[T]): Dataset[T]</code>
<code>unionByName</code>	<code>unionByName(other: Dataset[T]): Dataset[T]</code>
<code>where</code>	<pre> where(condition: Column): Dataset[T] where(conditionExpr: String): Dataset[T] </pre>

as Typed Transformation

```

as(alias: String): Dataset[T]
as(alias: Symbol): Dataset[T]

```

```
as ...FIXME
```

Enforcing Type — as Typed Transformation

```
as[U: Encoder]: Dataset[U]
```

`as[T]` allows for converting from a weakly-typed `Dataset` of `Rows` to `Dataset[T]` with `T` being a domain class (that can enforce a stronger schema).

```
// Create DataFrame of pairs
val df = Seq("hello", "world!").zipWithIndex.map(_.swap).toDF("id", "token")

scala> df.printSchema
root
|-- id: integer (nullable = false)
|-- token: string (nullable = true)

scala> val ds = df.as[(Int, String)]
ds: org.apache.spark.sql.Dataset[(Int, String)] = [id: int, token: string]

// It's more helpful to have a case class for the conversion
final case class MyRecord(id: Int, token: String)

scala> val myRecords = df.as[MyRecord]
myRecords: org.apache.spark.sql.Dataset[MyRecord] = [id: int, token: string]
```

Repartitioning Dataset with Shuffle Disabled — coalesce Typed Transformation

```
coalesce(numPartitions: Int): Dataset[T]
```

`coalesce` operator repartitions the `dataset` to exactly `numPartitions` partitions.

Internally, `coalesce` creates a `Repartition` logical operator with `shuffle` disabled (which is marked as `false` in the below `explain`'s output).

```

scala> spark.range(5).coalesce(1).explain(extended = true)
== Parsed Logical Plan ==
Repartition 1, false
+- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
Repartition 1, false
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Repartition 1, false
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
Coalesce 1
+- *Range (0, 5, step=1, splits=Some(8))

```

dropDuplicates Typed Transformation

```

dropDuplicates(): Dataset[T]
dropDuplicates(colNames: Array[String]): Dataset[T]
dropDuplicates(colNames: Seq[String]): Dataset[T]
dropDuplicates(col1: String, cols: String*): Dataset[T]

```

dropDuplicates ...FIXME

except Typed Transformation

```
except(other: Dataset[T]): Dataset[T]
```

except ...FIXME

exceptAll Typed Transformation

```
exceptAll(other: Dataset[T]): Dataset[T]
```

exceptAll ...FIXME

filter Typed Transformation

```
filter(condition: Column): Dataset[T]
filter(conditionExpr: String): Dataset[T]
filter(func: T => Boolean): Dataset[T]
```

`filter ...FIXME`

Creating Zero or More Records — `flatMap` Typed Transformation

```
flatMap[U: Encoder](func: T => TraversableOnce[U]): Dataset[U]
```

`flatMap` returns a new `Dataset` (of type `U`) with all records (of type `T`) mapped over using the function `func` and then flattening the results.

Note

`flatMap` can create new records. It deprecated `explode`.

```
final case class Sentence(id: Long, text: String)
val sentences = Seq(Sentence(0, "hello world"), Sentence(1, "witaj swiecie")).toDS

scala> sentences.flatMap(s => s.text.split("\\s+")).show
+-----+
| value|
+-----+
| hello|
| world|
| witaj|
| swiecie|
+-----+
```

Internally, `flatMap` calls `mapPartitions` with the partitions `flatMap(ped)`.

`intersect` Typed Transformation

```
intersect(other: Dataset[T]): Dataset[T]
```

`intersect ...FIXME`

`intersectAll` Typed Transformation

```
intersectAll(other: Dataset[T]): Dataset[T]
```

```
intersectAll ...FIXME
```

joinWith Typed Transformation

```
joinWith[U](other: Dataset[U], condition: Column): Dataset[(T, U)]  
joinWith[U](other: Dataset[U], condition: Column, joinType: String): Dataset[(T, U)]
```

```
joinWith ...FIXME
```

limit Typed Transformation

```
limit(n: Int): Dataset[T]
```

```
limit ...FIXME
```

map Typed Transformation

```
map[U : Encoder](func: T => U): Dataset[U]
```

```
map ...FIXME
```

mapPartitions Typed Transformation

```
mapPartitions[U : Encoder](func: Iterator[T] => Iterator[U]): Dataset[U]
```

```
mapPartitions ...FIXME
```

Randomly Split Dataset Into Two or More Datasets Per Weight — randomSplit Typed Transformation

```
randomSplit(weights: Array[Double]): Array[Dataset[T]]  
randomSplit(weights: Array[Double], seed: Long): Array[Dataset[T]]
```

`randomSplit` randomly splits the `Dataset` per `weights`.

`weights` doubles should sum up to `1` and will be normalized if they do not.

You can define `seed` and if you don't, a random `seed` will be used.

Note

`randomSplit` is commonly used in Spark MLlib to split an input Dataset into two datasets for training and validation.

```
val ds = spark.range(10)
scala> ds.randomSplit(Array[Double](2, 3)).foreach(_.show)
+---+
| id|
+---+
|  0|
|  1|
|  2|
+---+
+---+
| id|
+---+
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
+---+
```

Repartitioning Dataset (Shuffle Enabled) — `repartition`

Typed Transformation

```
repartition(partitionExprs: Column*): Dataset[T]
repartition(numPartitions: Int): Dataset[T]
repartition(numPartitions: Int, partitionExprs: Column*): Dataset[T]
```

`repartition` operators repartition the `Dataset` to exactly `numPartitions` partitions or using `partitionExprs` expressions.

Internally, `repartition` creates a [Repartition](#) or [RepartitionByExpression](#) logical operators with `shuffle` enabled (which is `true` in the below `explain`'s output beside `Repartition`).

```

scala> spark.range(5).repartition(1).explain(extended = true)
== Parsed Logical Plan ==
Repartition 1, true
+- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
Repartition 1, true
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Repartition 1, true
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
Exchange RoundRobinPartitioning(1)
+- *Range (0, 5, step=1, splits=Some(8))

```

Note

`repartition` methods correspond to SQL's **DISTRIBUTE BY** or **CLUSTER BY clauses**.

repartitionByRange Typed Transformation

```

repartitionByRange(partitionExprs: Column*): Dataset[T] (1)
repartitionByRange(numPartitions: Int, partitionExprs: Column*): Dataset[T]

```

1. Uses `spark.sql.shuffle.partitions` configuration property for the number of partitions to use

`repartitionByRange` simply creates a `Dataset` with a `RepartitionByExpression` logical operator.

```

scala> spark.version
res1: String = 2.3.1

val q = spark.range(10).repartitionByRange(numPartitions = 5, $"id")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'RepartitionByExpression ['id ASC NULLS FIRST], 5
01 +- AnalysisBarrier
02     +- Range (0, 10, step=1, splits=Some(8))

scala> println(q.queryExecution.toRdd.getNumPartitions)
5

scala> println(q.queryExecution.toRdd.toDebugString)
(5) ShuffledRowRDD[18] at toRdd at <console>:26 []
+- (8) MapPartitionsRDD[17] at toRdd at <console>:26 []
  |  MapPartitionsRDD[13] at toRdd at <console>:26 []
  |  MapPartitionsRDD[12] at toRdd at <console>:26 []
  |  ParallelCollectionRDD[11] at toRdd at <console>:26 []

```

`repartitionByRange` uses a `SortOrder` with the `Ascending` sort order, i.e. *ascending nulls first*, when no explicit sort order is specified.

`repartitionByRange` throws a `IllegalArgumentException` when no `partitionExprs` partition-by expression is specified.

```
requirement failed: At least one partition-by expression must be specified.
```

sample Typed Transformation

```

sample(withReplacement: Boolean, fraction: Double): Dataset[T]
sample(withReplacement: Boolean, fraction: Double, seed: Long): Dataset[T]
sample(fraction: Double): Dataset[T]
sample(fraction: Double, seed: Long): Dataset[T]

```

```
sample ...FIXME
```

select Typed Transformation

```

select[U1](c1: TypedColumn[T, U1]): Dataset[U1]
select[U1, U2](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2]): Dataset[(U1, U2)]
select[U1, U2, U3](
  c1: TypedColumn[T, U1],
  c2: TypedColumn[T, U2],
  c3: TypedColumn[T, U3]): Dataset[(U1, U2, U3)]
select[U1, U2, U3, U4](
  c1: TypedColumn[T, U1],
  c2: TypedColumn[T, U2],
  c3: TypedColumn[T, U3],
  c4: TypedColumn[T, U4]): Dataset[(U1, U2, U3, U4)]
select[U1, U2, U3, U4, U5](
  c1: TypedColumn[T, U1],
  c2: TypedColumn[T, U2],
  c3: TypedColumn[T, U3],
  c4: TypedColumn[T, U4],
  c5: TypedColumn[T, U5]): Dataset[(U1, U2, U3, U4, U5)]

```

`select ...FIXME`

sort Typed Transformation

```

sort(sortExprs: Column*): Dataset[T]
sort(sortCol: String, sortCols: String*): Dataset[T]

```

`sort ...FIXME`

sortWithinPartitions Typed Transformation

```

sortWithinPartitions(sortExprs: Column*): Dataset[T]
sortWithinPartitions(sortCol: String, sortCols: String*): Dataset[T]

```

`sortWithinPartitions` simply calls the internal `sortInternal` method with the `global` flag disabled (`false`).

toJSON Typed Transformation

```
toJSON: Dataset[String]
```

`toJSON` maps the content of `Dataset` to a `Dataset` of strings in JSON format.

```
scala> val ds = Seq("hello", "world", "foo bar").toDS
ds: org.apache.spark.sql.Dataset[String] = [value: string]

scala> ds.toJSON.show
+-----+
|       value|
+-----+
| {"value":"hello"}|
| {"value":"world"}|
| {"value":"foo bar"}|
+-----+
```

Internally, `toJSON` grabs the `RDD[InternalRow]` (of the `QueryExecution` of the `Dataset`) and maps the records (per RDD partition) into JSON.

Note `toJSON` uses Jackson's JSON parser—[jackson-module-scala](#).

Transforming Datasets — `transform` Typed Transformation

```
transform[U](t: Dataset[T] => Dataset[U]): Dataset[U]
```

`transform` applies `t` function to the source `Dataset[T]` to produce a result `Dataset[U]`. It is for chaining custom transformations.

```
val dataset = spark.range(5)

// Transformation t
import org.apache.spark.sql.Dataset
def withDoubled(longs: Dataset[java.lang.Long]) = longs.withColumn("doubled", 'id * 2)

scala> dataset.transform(withDoubled).show
+---+-----+
| id|doubled|
+---+-----+
|  0|      0|
|  1|      2|
|  2|      4|
|  3|      6|
|  4|      8|
+---+-----+
```

Internally, `transform` executes `t` function on the current `Dataset[T]`.

union Typed Transformation

```
union(other: Dataset[T]): Dataset[T]
```

`union` ...FIXME

unionByName Typed Transformation

```
unionByName(other: Dataset[T]): Dataset[T]
```

`unionByName` creates a new `Dataset` that is an union of the rows in this and the other Datasets column-wise, i.e. the order of columns in Datasets does not matter as long as their names and number match.

```
val left = spark.range(1).withColumn("rand", rand()).select("id", "rand")
val right = Seq(("0.1", 11)).toDF("rand", "id")
val q = left.unionByName(right)
scala> q.show
+---+-----+
| id|      rand|
+---+-----+
|  0|0.14747380134150134|
| 11|      0.1|
+---+-----+
```

Internally, `unionByName` creates a [Union](#) logical operator for this `Dataset` and [Project](#) logical operator with the `other` `Dataset`.

In the end, `unionByName` applies the [CombineUnions](#) logical optimization to the `Union` logical operator and requests the result `LogicalPlan` to [wrap the child operators](#) with [AnalysisBarriers](#).

```
scala> println(q.queryExecution.logical.numberedTreeString)
00 'Union
01 :- AnalysisBarrier
02 :   +- Project [id#90L, rand#92]
03 :     +- Project [id#90L, rand(-9144575865446031058) AS rand#92]
04 :       +- Range (0, 1, step=1, splits=Some(8))
05 +- AnalysisBarrier
06   +- Project [id#103, rand#102]
07     +- Project [_1#99 AS rand#102, _2#100 AS id#103]
08       +- LocalRelation [_1#99, _2#100]
```

`unionByName` throws an `AnalysisException` if there are duplicate columns in either Dataset.

```
Found duplicate column(s)
```

`unionByName` throws an `AnalysisException` if there are columns in this Dataset has a column that is not available in the `other` Dataset.

```
Cannot resolve column name "[name]" among ([rightNames])
```

where Typed Transformation

```
where(condition: Column): Dataset[T]
where(conditionExpr: String): Dataset[T]
```

`where` is simply a synonym of the `filter` operator, i.e. passes the input parameters along to `filter`.

Creating Streaming Dataset with EventTimeWatermark Logical Operator — withWatermark Streaming Typed Transformation

```
withWatermark(eventTime: String, delayThreshold: String): Dataset[T]
```

Internally, `withWatermark` creates a `Dataset` with `EventTimeWatermark` logical plan for [streaming Datasets](#).

Note	<code>withWatermark</code> uses <code>EliminateEventTimeWatermark</code> logical rule to eliminate <code>EventTimeWatermark</code> logical plan for non-streaming batch <code>Datasets</code> .
------	---

```
// Create a batch dataset
val events = spark.range(0, 50, 10).
  withColumn("timestamp", from_unixtime(unix_timestamp - 'id)).
  select('timestamp, 'id as "count")
scala> events.show
+-----+-----+
|      timestamp|count|
+-----+-----+
|2017-06-25 21:21:14|    0|
|2017-06-25 21:21:04|   10|
|2017-06-25 21:20:54|   20|
|2017-06-25 21:20:44|   30|
|2017-06-25 21:20:34|   40|
+-----+-----+

// the dataset is a non-streaming batch one...
scala> events.isStreaming
res1: Boolean = false

// ...so EventTimeWatermark is not included in the logical plan
val watermarked = events.
  withWatermark(eventTime = "timestamp", delayThreshold = "20 seconds")
scala> println(watermarked.queryExecution.logical.numberedTreeString)
00 Project [timestamp#284, id#281L AS count#288L]
01 +- Project [id#281L, from_unixtime((unix_timestamp(current_timestamp()), yyyy-MM-dd
HH:mm:ss, Some(America/Chicago)) - id#281L), yyyy-MM-dd HH:mm:ss, Some(America/Chicago
)) AS timestamp#284]
02     +- Range (0, 50, step=10, splits=Some(8))

// Let's create a streaming Dataset
import org.apache.spark.sql.types.StructType
val schema = new StructType().
  add($"timestamp".timestamp).
  add($"count".long)
scala> schema.printTreeString
root
|-- timestamp: timestamp (nullable = true)
|-- count: long (nullable = true)

val events = spark.
  readStream.
  schema(schema).
  csv("events").
  withWatermark(eventTime = "timestamp", delayThreshold = "20 seconds")
scala> println(events.queryExecution.logical.numberedTreeString)
00 'EventTimeWatermark 'timestamp, interval 20 seconds
01 +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@75abccdd4, csv, List
(), Some(StructType(StructField(timestamp, TimestampType, true), StructField(count, LongTy
pe, true))), List(), None, Map(path -> events), None), FileSource[events], [timestamp#329,
count#330L]
```

	<p><code>delayThreshold</code> is parsed using <code>calendarInterval.fromString</code> with interval formal described in TimeWindow unary expression.</p>
Note	<pre>0 years 0 months 1 week 0 days 0 hours 1 minute 20 seconds 0 milliseconds 0 microseconds 0 nanoseconds</pre>
Note	<p><code>delayThreshold</code> must not be negative (and <code>milliseconds</code> and <code>months</code> should both be equal or greater than <code>0</code>).</p>
Note	<p><code>withWatermark</code> is used when...FIXME</p>

Dataset API—Untyped Transformations

Untyped transformations are part of the Dataset API for transforming a `Dataset` to a `DataFrame`, a `Column`, a `RelationalGroupedDataset`, a `DataFrameNaFunctions` or a `DataFrameStatFunctions` (and hence *untyped*).

Note

Untyped transformations are the methods in the `Dataset` Scala class that are grouped in `untypedrel` group name, i.e. `@group untypedrel`.

Table 1. Dataset API's Untyped Transformations

Transformation	Description
<code>agg</code>	<code>agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame</code> <code>agg(expr: Column, exprs: Column*): DataFrame</code> <code>agg(exprs: Map[String, String]): DataFrame</code>
<code>apply</code>	Selects a column based on the column name (i.e. maps a <code>Dataset</code> to a <code>Column</code>) <code>apply(colName: String): Column</code>
<code>col</code>	Selects a column based on the column name (i.e. maps a <code>Dataset</code> to a <code>Column</code>) <code>col(colName: String): Column</code>
<code>colRegex</code>	<code>colRegex(colName: String): Column</code> Selects a column based on the column name specified as a regex (i.e. maps a <code>Dataset</code> onto a <code>Column</code>)
<code>crossJoin</code>	<code>crossJoin(right: Dataset[_]): DataFrame</code>
<code>cube</code>	<code>cube(cols: Column*): RelationalGroupedDataset</code> <code>cube(col1: String, cols: String*): RelationalGroupedDataset</code>
<code>drop</code>	<code>drop(colName: String): DataFrame</code> <code>drop(colNames: String*): DataFrame</code> <code>drop(col: Column): DataFrame</code>

groupBy	groupBy(cols: Column*): RelationalGroupedDataset groupBy(col1: String, cols: String*): RelationalGroupedDataset
join	join(right: Dataset[_]): DataFrame join(right: Dataset[_], usingColumn: String): DataFrame join(right: Dataset[_], usingColumns: Seq[String]): DataFrame join(right: Dataset[_], usingColumns: Seq[String], joinType: String): DataFrame join(right: Dataset[_], joinExprs: Column): DataFrame join(right: Dataset[_], joinExprs: Column, joinType: String): DataFrame
na	na: DataFrameNaFunctions
rollup	rollup(cols: Column*): RelationalGroupedDataset rollup(col1: String, cols: String*): RelationalGroupedDataset
select	select(cols: Column*): DataFrame select(col: String, cols: String*): DataFrame
selectExpr	selectExpr(exprs: String*): DataFrame
stat	stat: DataFrameStatFunctions
withColumn	withColumn(colName: String, col: Column): DataFrame
withColumnRenamed	withColumnRenamed(existingName: String, newName: String): DataFrame

agg Untyped Transformation

```
agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame
agg(expr: Column, exprs: Column*): DataFrame
agg(exprs: Map[String, String]): DataFrame
```

```
agg ...FIXME
```

apply Untyped Transformation

```
apply(colName: String): Column
```

`apply` selects a column based on the column name (i.e. maps a `Dataset` onto a `Column`).

col Untyped Transformation

```
col(colName: String): Column
```

`col` selects a column based on the column name (i.e. maps a `Dataset` onto a `Column`).

Internally, `col` branches off per the input column name.

If the column name is `*` (a star), `col` simply creates a [Column](#) with [ResolvedStar](#) expression (with the [schema output attributes](#) of the [analyzed logical plan](#) of the [QueryExecution](#)).

Otherwise, `col` uses [colRegex](#) untyped transformation when [spark.sql.parser.quotedRegexColumnNames](#) configuration property is enabled.

In the case when the column name is not `*` and [spark.sql.parser.quotedRegexColumnNames](#) configuration property is disabled, `col` creates a [Column](#) with the column name [resolved](#) (as a [NamedExpression](#)).

colRegex Untyped Transformation

```
colRegex(colName: String): Column
```

`colRegex` selects a column based on the column name specified as a regex (i.e. maps a `Dataset` onto a `Column`).

Note

`colRegex` is used in `col` when [spark.sql.parser.quotedRegexColumnNames](#) configuration property is enabled (and the column name is not `*`).

Internally, `colRegex` matches the input column name to different regular expressions (in the order):

1. For column names with quotes without a qualifier, `colRegex` simply creates a [Column](#) with a [UnresolvedRegex](#) (with no table)

2. For column names with quotes with a qualifier, `colRegex` simply creates a [Column](#) with a [UnresolvedRegex](#) (with a table specified)
3. For other column names, `colRegex` (behaves like `col` and) creates a [Column](#) with the column name [resolved](#) (as a [NamedExpression](#))

crossJoin Untyped Transformation

```
crossJoin(right: Dataset[_]): DataFrame
```

`crossJoin` ...FIXME

cube Untyped Transformation

```
cube(cols: Column*): RelationalGroupedDataset  
cube(col1: String, cols: String*): RelationalGroupedDataset
```

`cube` ...FIXME

Dropping One or More Columns — drop Untyped Transformation

```
drop(colName: String): DataFrame  
drop(colNames: String*): DataFrame  
drop(col: Column): DataFrame
```

`drop` ...FIXME

groupBy Untyped Transformation

```
groupBy(cols: Column*): RelationalGroupedDataset  
groupBy(col1: String, cols: String*): RelationalGroupedDataset
```

`groupBy` ...FIXME

join Untyped Transformation

```
join(right: Dataset[_]): DataFrame  
join(right: Dataset[_], usingColumn: String): DataFrame  
join(right: Dataset[_], usingColumns: Seq[String]): DataFrame  
join(right: Dataset[_], usingColumns: Seq[String], joinType: String): DataFrame  
join(right: Dataset[_], joinExprs: Column): DataFrame  
join(right: Dataset[_], joinExprs: Column, joinType: String): DataFrame
```

join ...FIXME

na Untyped Transformation

```
na: DataFrameNaFunctions
```

na simply creates a [DataFrameNaFunctions](#) to work with missing data.

rollup Untyped Transformation

```
rollup(cols: Column*): RelationalGroupedDataset  
rollup(col1: String, cols: String*): RelationalGroupedDataset
```

rollup ...FIXME

select Untyped Transformation

```
select(cols: Column*): DataFrame  
select(col: String, cols: String*): DataFrame
```

select ...FIXME

Projecting Columns using SQL Statements — selectExpr Untyped Transformation

```
selectExpr(exprs: String*): DataFrame
```

selectExpr is like select , but accepts SQL statements.

```
val ds = spark.range(5)

scala> ds.selectExpr("rand() as random").show
16/04/14 23:16:06 INFO HiveSqlParser: Parsing command: rand() as random
+-----+
|      random|
+-----+
| 0.887675894185651|
| 0.36766085091074086|
| 0.2700020856675186|
| 0.1489033635529543|
| 0.5862990791950973|
+-----+
```

Internally, it executes `select` with every expression in `exprs` mapped to [Column](#) (using [SparkSqlParser.parseExpression](#)).

```
scala> ds.select(expr("rand() as random")).show
+-----+
|      random|
+-----+
| 0.5514319279894851|
| 0.2876221510433741|
| 0.4599999092045741|
| 0.5708558868374893|
| 0.6223314406247136|
+-----+
```

stat Untyped Transformation

```
stat: DataFrameStatFunctions
```

`stat` simply creates a [DataFrameStatFunctions](#) to work with statistic functions.

withColumn Untyped Transformation

```
withColumn(colName: String, col: Column): DataFrame
```

```
withColumn ...FIXME
```

withColumnRenamed Untyped Transformation

```
withColumnRenamed(existingName: String, newName: String): DataFrame
```

```
withColumnRenamed ...FIXME
```

Dataset API—Basic Actions

Basic actions are a group of operators (*methods*) of the [Dataset API](#) for transforming a `Dataset` into a session-scoped or global temporary view and *other basic actions* (FIXME).

Note

Basic actions are the methods in the `Dataset` Scala class that are grouped in `basic` group name, i.e. `@group basic`.

Table 1. Dataset API's Basic Actions

Action	Description
<code>cache</code>	<pre>cache(): this.type</pre> <p>Marks the <code>Dataset</code> to be persisted (<i>cached</i>) and is actually a synonym of persist basic action</p>
<code>checkpoint</code>	<pre>checkpoint(): Dataset[T] checkpoint(eager: Boolean): Dataset[T]</pre> <p>Checkpoints the <code>Dataset</code> in a reliable way (using a reliable HDFS-compliant file system, e.g. Hadoop HC or Amazon S3)</p>
<code>columns</code>	<pre>columns: Array[String]</pre>
<code>createGlobalTempView</code>	<pre>createGlobalTempView(viewName: String): Unit</pre>
<code>createOrReplaceGlobalTempView</code>	<pre>createOrReplaceGlobalTempView(viewName: String): Unit</pre>
<code>createOrReplaceTempView</code>	<pre>createOrReplaceTempView(viewName: String): Unit</pre>
<code>createView</code>	<pre>createView(viewName: String): Unit</pre>
<code>dtypes</code>	<pre>dtypes: Array[(String, String)]</pre>

	<code>explain(): Unit</code> <code>explain(extended: Boolean): Unit</code>		
explain	Displays the logical and physical plans of the <code>Dataset</code> i.e. displays the logical and physical plans (with option cost and codegen summaries) to the standard output		
hint	<code>hint(name: String, parameters: Any*): Dataset[T]</code>		
inputFiles	<code>inputFiles: Array[String]</code>		
isEmpty	<code>isEmpty: Boolean</code> (New in 2.4.4)		
isLocal	<code>isLocal: Boolean</code>		
localCheckpoint	<code>localCheckpoint(): Dataset[T]</code> <code>localCheckpoint(eager: Boolean): Dataset[T]</code> Checkpoints the <code>Dataset</code> locally on executors (and therefore unreliable)		
persist	<code>persist(): this.type (1)</code> <code>persist(newLevel: StorageLevel): this.type</code> 1. Assumes the default storage level <code>MEMORY_AND_DISK</code> Marks the <code>Dataset</code> to be persisted the next time an action is executed Internally, <code>persist</code> simply request the <code>CacheManager</code> cache the structured query.		
	<table border="1"> <tr> <td>Note</td> <td><code>persist</code> uses the <code>CacheManager</code> from the <code>SharedState</code> associated with the <code>SparkSession</code> (of the <code>Dataset</code>).</td> </tr> </table>	Note	<code>persist</code> uses the <code>CacheManager</code> from the <code>SharedState</code> associated with the <code>SparkSession</code> (of the <code>Dataset</code>).
Note	<code>persist</code> uses the <code>CacheManager</code> from the <code>SharedState</code> associated with the <code>SparkSession</code> (of the <code>Dataset</code>).		
printSchema	<code>printSchema(): Unit</code>		

rdd	rdd: RDD[T]
schema	schema: StructType
storageLevel	storageLevel: StorageLevel
toDF	toDF(): DataFrame toDF(colNames: String*): DataFrame
unpersist	unpersist(): this.type unpersist(blocking: Boolean): this.type
	Unpersists the Dataset
write	write: DataFrameWriter[T] Returns a DataFrameWriter for saving the content of (non-streaming) Dataset out to an external storage

Reliably Checkpointing Dataset— checkpoint Basic Action

```
checkpoint(): Dataset[T] (1)
checkpoint(eager: Boolean): Dataset[T] (2)
```

1. eager and reliableCheckpoint flags enabled
2. reliableCheckpoint flag enabled

Note	checkpoint is an experimental operator and the API is evolving towards becoming stable.
------	---

checkpoint simply requests the Dataset to checkpoint with the given eager flag and the reliableCheckpoint flag enabled.

createTempView Basic Action

```
createTempView(viewName: String): Unit
```

createTempView ...FIXME

Note

createTempView is used when...FIXME

createOrReplaceTempView Basic Action

```
createOrReplaceTempView(viewName: String): Unit
```

createOrReplaceTempView ...FIXME

Note

createOrReplaceTempView is used when...FIXME

createGlobalTempView Basic Action

```
createGlobalTempView(viewName: String): Unit
```

createGlobalTempView ...FIXME

Note

createGlobalTempView is used when...FIXME

createOrReplaceGlobalTempView Basic Action

```
createOrReplaceGlobalTempView(viewName: String): Unit
```

createOrReplaceGlobalTempView ...FIXME

Note

createOrReplaceGlobalTempView is used when...FIXME

createTempViewCommand Internal Method

```
createTempViewCommand(  
  viewName: String,  
  replace: Boolean,  
  global: Boolean): CreateViewCommand
```

createTempViewCommand ...FIXME

Note	<code>createTempViewCommand</code> is used when the following <code>Dataset</code> operators are used: <code>Dataset.createTempView</code> , <code>Dataset.createOrReplaceTempView</code> , <code>Dataset.createGlobalTempView</code> and <code>Dataset.createOrReplaceGlobalTempView</code> .
------	--

Displaying Logical and Physical Plans, Their Cost and Codegen — `explain` Basic Action

```
explain(): Unit (1)
explain(extended: Boolean): Unit
```

1. Turns the `extended` flag on

`explain` prints the `logical` and (with `extended` flag enabled) `physical` plans, their cost and codegen to the console.

Tip	Use <code>explain</code> to review the structured queries and optimizations applied.
-----	--

Internally, `explain` creates a `ExplainCommand` logical command and requests `SessionState` to `execute it` (to get a `QueryExecution` back).

Note	<code>explain</code> uses <code>ExplainCommand</code> logical command that, when <code>executed</code> , gives different text representations of <code>QueryExecution</code> (for the Dataset's <code>LogicalPlan</code>) depending on the flags (e.g. <code>extended</code> , <code>codegen</code> , and <code>cost</code> which are disabled by default).
------	--

`explain` then requests `QueryExecution` for the optimized physical query plan and collects the records (as `InternalRow` objects).

Note	<code>explain</code> uses Dataset's <code>SparkSession</code> to access the current <code>SessionState</code> .
------	---

In the end, `explain` goes over the `InternalRow` records and converts them to lines to display to console.

Note	<code>explain</code> "converts" an <code>InternalRow</code> record to a line using <code>getString</code> at position <code>0</code> .
------	--

Tip	If you are serious about query debugging you could also use the Debugging Query Execution facility .
-----	--

```
scala> spark.range(10).explain(extended = true)
== Parsed Logical Plan ==
Range (0, 10, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
Range (0, 10, step=1, splits=Some(8))

== Optimized Logical Plan ==
Range (0, 10, step=1, splits=Some(8))

== Physical Plan ==
*Range (0, 10, step=1, splits=Some(8))
```

Specifying Hint— `hint` Basic Action

```
hint(name: String, parameters: Any*): Dataset[T]
```

`hint` operator is part of [Hint Framework](#) to specify a `hint` (by `name` and `parameters`) for a `Dataset`.

Internally, `hint` simply attaches [UnresolvedHint](#) unary logical operator to an "analyzed" `Dataset` (i.e. the [analyzed logical plan](#) of a `dataset`).

```
val ds = spark.range(3)
val plan = ds.queryExecution.logical
scala> println(plan.numberedTreeString)
00 Range (0, 3, step=1, splits=Some(8))

// Attach a hint
val dsHinted = ds_hint("myHint", 100, true)
val plan = dsHinted.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- Range (0, 3, step=1, splits=Some(8))
```

Note

`hint` adds an [UnresolvedHint](#) unary logical operator to an analyzed logical plan that indirectly triggers [analysis phase](#) that executes [logical commands](#) and their unions as well as resolves all hints that have already been added to a logical plan.

```
// FIXME Demo with UnresolvedHint
```

Locally Checkpointing Dataset — `localCheckpoint` Basic Action

```
localCheckpoint(): Dataset[T] (1)
localCheckpoint(eager: Boolean): Dataset[T]
```

1. `eager` flag enabled

`localCheckpoint` simply uses `Dataset.checkpoint` operator with the input `eager` flag and `reliableCheckpoint` flag disabled (`false`).

checkpoint Internal Method

```
checkpoint(eager: Boolean, reliableCheckpoint: Boolean): Dataset[T]
```

`checkpoint` requests `QueryExecution` (of the `Dataset`) to generate an RDD of internal `binary rows` (aka `internalRdd`) and then requests the RDD to make a copy of all the rows (by adding a `MapPartitionsRDD`).

Depending on `reliableCheckpoint` flag, `checkpoint` marks the RDD for (reliable) checkpointing (`true`) or local checkpointing (`false`).

With `eager` flag on, `checkpoint` counts the number of records in the RDD (by executing `RDD.count`) that gives the effect of immediate eager checkpointing.

`checkpoint` requests `QueryExecution` (of the `Dataset`) for optimized physical query plan (the plan is used to get the `outputPartitioning` and `outputOrdering` for the result `Dataset`).

In the end, `checkpoint` creates a `DataFrame` with a new logical plan node for scanning data from an RDD of `InternalRows` (`LogicalRDD`).

Note	<code>checkpoint</code> is used in the <code>Dataset</code> untyped transformations, i.e. <code>checkpoint</code> and <code>localCheckpoint</code> .
------	--

Generating RDD of Internal Binary Rows — `rdd` Basic Action

```
rdd: RDD[T]
```

Whenever you are in need to convert a `Dataset` into a `RDD`, executing `rdd` method gives you the `RDD` of the proper input object type (not [Row as in DataFrames](#)) that sits behind the `Dataset`.

```
scala> val rdd = tokens.rdd
rdd: org.apache.spark.rdd.RDD[Token] = MapPartitionsRDD[11] at rdd at <console>:30
```

Internally, it looks [ExpressionEncoder](#) (for the `Dataset`) up and accesses the `deserializer` expression. That gives the `DataType` of the result of evaluating the expression.

Note	A deserializer expression is used to decode an InternalRow to an object of type <code>T</code> . See ExpressionEncoder .
------	--

It then executes a [deserializeToObject](#) logical operator that will produce a `RDD[InternalRow]` that is converted into the proper `RDD[T]` using the `DataType` and `T`.

Note	It is a lazy operation that "produces" a <code>RDD[T]</code> .
------	--

Accessing Schema — `schema` Basic Action

A `Dataset` has a **schema**.

```
schema: StructType
```

Tip	You may also use the following methods to learn about the schema:
-----	---

- `printSchema(): Unit`
- [explain](#)

Converting Typed Dataset to Untyped DataFrame — `toDF` Basic Action

```
toDF(): DataFrame
toDF(colNames: String*): DataFrame
```

`toDF` converts a `Dataset` into a `DataFrame`.

Internally, the empty-argument `toDF` creates a `Dataset[Row]` using the `Dataset`'s [SparkSession](#) and [QueryExecution](#) with the encoder being [RowEncoder](#).

Caution	FIXME Describe <code>toDF(colNames: String*)</code>
---------	---

Unpersisting Cached Dataset — `unpersist` Basic Action

```
unpersist(): this.type  
unpersist(blocking: Boolean): this.type
```

`unpersist` uncache the `Dataset` possibly by `blocking` the call.

Internally, `unpersist` requests `CacheManager` to [uncache the query](#).

Caution

FIXME

Accessing DataFrameWriter (to Describe Writing Dataset) — `write` Basic Action

```
write: DataFrameWriter[T]
```

`write` gives [DataFrameWriter](#) for records of type `T`.

```
import org.apache.spark.sql.{DataFrameWriter, Dataset}  
val ints: Dataset[Int] = (0 to 5).toDS  
val writer: DataFrameWriter[Int] = ints.write
```

`isEmpty` Typed Transformation

```
isEmpty: Boolean
```

`isEmpty` ...FIXME

`isLocal` Typed Transformation

```
isLocal: Boolean
```

`isLocal` ...FIXME

Dataset API—Actions

Actions are part of the [Dataset API](#) for...FIXME

Note	Actions are the methods in the <code>Dataset</code> Scala class that are grouped in <code>action</code> group name, i.e. <code>@group action</code> .
------	---

Table 1. Dataset API's Actions

Action	Description
<code>collect</code>	<code>collect(): Array[T]</code>
<code>count</code>	<code>count(): Long</code>
<code>describe</code>	<code>describe(cols: String*): DataFrame</code>
<code>first</code>	<code>first(): T</code>
<code>foreach</code>	<code>foreach(f: T => Unit): Unit</code>
<code>foreachPartition</code>	<code>foreachPartition(f: Iterator[T] => Unit): Unit</code>
<code>head</code>	<code>head(): T</code> <code>head(n: Int): Array[T]</code>
<code>reduce</code>	<code>reduce(func: (T, T) => T): T</code>
<code>show</code>	<code>show(): Unit</code> <code>show(truncate: Boolean): Unit</code> <code>show numRows: Int): Unit</code> <code>show numRows: Int, truncate: Boolean): Unit</code> <code>show numRows: Int, truncate: Int): Unit</code> <code>show numRows: Int, truncate: Int, vertical: Boolean): Unit</code>

	Computes specified statistics for numeric and string columns. The default statistics are: count , mean , stddev , min , max and 25% , 50% , 75% percentiles.
summary	<pre>summary(statistics: String*): DataFrame</pre> <p>Note <code>summary</code> is an extended version of the <code>describe</code> action that simply calculates count , mean , stddev , min and max statistics.</p>
take	<code>take(n: Int): Array[T]</code>
toLocalIterator	<code>toLocalIterator(): java.util.Iterator[T]</code>

collect Action

```
collect(): Array[T]
```

```
collect ...FIXME
```

count Action

```
count(): Long
```

```
count ...FIXME
```

Calculating Basic Statistics — describe Action

```
describe(cols: String*): DataFrame
```

```
describe ...FIXME
```

first Action

```
first(): T
```

```
first ...FIXME
```

foreach Action

```
foreach(f: T => Unit): Unit
```

```
foreach ...FIXME
```

foreachPartition Action

```
foreachPartition(f: Iterator[T] => Unit): Unit
```

```
foreachPartition ...FIXME
```

head Action

```
head(): T (1)  
head(n: Int): Array[T]
```

1. Calls the other `head` with `n` as `1` and takes the first element

```
head ...FIXME
```

reduce Action

```
reduce(func: (T, T) => T): T
```

```
reduce ...FIXME
```

show Action

```
show(): Unit  
show(truncate: Boolean): Unit  
show numRows: Int): Unit  
show(numRows: Int, truncate: Boolean): Unit  
show(numRows: Int, truncate: Int): Unit  
show(numRows: Int, truncate: Int, vertical: Boolean): Unit
```

```
show ...FIXME
```

Calculating Statistics — `summary` Action

```
summary(statistics: String*): DataFrame
```

`summary` calculates specified statistics for numeric and string columns.

The default statistics are: `count` , `mean` , `stddev` , `min` , `max` and `25%` , `50%` , `75%` percentiles.

Note	<code>summary</code> accepts arbitrary approximate percentiles specified as a percentage (e.g. <code>10%</code>).
------	--

Internally, `summary` uses the `StatFunctions` to calculate the requested summaries for the [Dataset](#).

Taking First Records — `take` Action

```
take(n: Int): Array[T]
```

`take` is an action on a `Dataset` that returns a collection of `n` records.

Warning	<code>take</code> loads all the data into the memory of the Spark application's driver process and for a large <code>n</code> could result in <code>OutOfMemoryError</code> .
---------	---

Internally, `take` creates a new `Dataset` with `Limit` logical plan for `Literal` expression and the current `LogicalPlan` . It then runs the [SparkPlan](#) that produces a `Array[InternalRow]` that is in turn decoded to `Array[T]` using a bounded [encoder](#).

`toLocalIterator` Action

```
toLocalIterator(): java.util.Iterator[T]
```

`toLocalIterator` ...FIXME

DataFrameNaFunctions — Working With Missing Data

`DataFrameNaFunctions` is used to work with [missing data](#) in a structured query (a [DataFrame](#)).

Table 1. DataFrameNaFunctions API

Method	Description
<code>drop</code>	<pre>drop(): DataFrame drop(cols: Array[String]): DataFrame drop(minNonNulls: Int): DataFrame drop(minNonNulls: Int, cols: Array[String]): DataFrame drop(minNonNulls: Int, cols: Seq[String]): DataFrame drop(cols: Seq[String]): DataFrame drop(how: String): DataFrame drop(how: String, cols: Array[String]): DataFrame drop(how: String, cols: Seq[String]): DataFrame</pre>
<code>fill</code>	<pre>fill(value: Boolean): DataFrame fill(value: Boolean, cols: Array[String]): DataFrame fill(value: Boolean, cols: Seq[String]): DataFrame fill(value: Double): DataFrame fill(value: Double, cols: Array[String]): DataFrame fill(value: Double, cols: Seq[String]): DataFrame fill(value: Long): DataFrame fill(value: Long, cols: Array[String]): DataFrame fill(value: Long, cols: Seq[String]): DataFrame fill(valueMap: Map[String, Any]): DataFrame fill(value: String): DataFrame fill(value: String, cols: Array[String]): DataFrame fill(value: String, cols: Seq[String]): DataFrame</pre>
<code>replace</code>	<pre>replace[T](cols: Seq[String], replacement: Map[T, T]): DataFrame replace[T](col: String, replacement: Map[T, T]): DataFrame</pre>

`DataFrameNaFunctions` is available using `na` untyped transformation.

```
val q: DataFrame = ...
q.na
```

convertToDouble Internal Method

```
convertToDouble(v: Any): Double
```

`convert.ToDouble ...FIXME`

Note

`convert.ToDouble` is used when...FIXME

drop Method

```
drop(): DataFrame
drop(cols: Array[String]): DataFrame
drop(minNonNulls: Int): DataFrame
drop(minNonNulls: Int, cols: Array[String]): DataFrame
drop(minNonNulls: Int, cols: Seq[String]): DataFrame
drop(cols: Seq[String]): DataFrame
drop(how: String): DataFrame
drop(how: String, cols: Array[String]): DataFrame
drop(how: String, cols: Seq[String]): DataFrame
```

`drop ...FIXME`

fill Method

```
fill(value: Boolean): DataFrame
fill(value: Boolean, cols: Array[String]): DataFrame
fill(value: Boolean, cols: Seq[String]): DataFrame
fill(value: Double): DataFrame
fill(value: Double, cols: Array[String]): DataFrame
fill(value: Double, cols: Seq[String]): DataFrame
fill(value: Long): DataFrame
fill(value: Long, cols: Array[String]): DataFrame
fill(value: Long, cols: Seq[String]): DataFrame
fill(valueMap: Map[String, Any]): DataFrame
fill(value: String): DataFrame
fill(value: String, cols: Array[String]): DataFrame
fill(value: String, cols: Seq[String]): DataFrame
```

`fill ...FIXME`

fillCol Internal Method

```
fillCol[T](col: StructField, replacement: T): Column
```

`fillCol ...FIXME`

Note

`fillCol` is used when...FIXME

fillMap Internal Method

```
fillMap(values: Seq[(String, Any)]): DataFrame
```

fillMap ...FIXME

Note	fillMap is used when...FIXME
------	------------------------------

fillValue Internal Method

```
fillValue[T](value: T, cols: Seq[String]): DataFrame
```

fillValue ...FIXME

Note	fillValue is used when...FIXME
------	--------------------------------

replace0 Internal Method

```
replace0[T](cols: Seq[String], replacement: Map[T, T]): DataFrame
```

replace0 ...FIXME

Note	replace0 is used when...FIXME
------	-------------------------------

replace Method

```
replace[T](cols: Seq[String], replacement: Map[T, T]): DataFrame  
replace[T](col: String, replacement: Map[T, T]): DataFrame
```

replace ...FIXME

replaceCol Internal Method

```
replaceCol(col: StructField, replacementMap: Map[_, _]): Column
```

replaceCol ...FIXME

Note	replaceCol is used when...FIXME
------	---------------------------------

DataFrameStatFunctions — Working With Statistic Functions

`DataFrameStatFunctions` is used to work with [statistic functions](#) in a structured query (a [DataFrame](#)).

Table 1. DataFrameStatFunctions API

Method	Description
approxQuantile	<pre>approxQuantile(cols: Array[String], probabilities: Array[Double], relativeError: Double): Array[Array[Double]] approxQuantile(col: String, probabilities: Array[Double], relativeError: Double): Array[Double]</pre>
bloomFilter	<pre>bloomFilter(col: Column, expectedNumItems: Long, fpp: Double): BloomFilter bloomFilter(col: Column, expectedNumItems: Long, numBits: Long): BloomFilter bloomFilter(colName: String, expectedNumItems: Long, fpp: Double): BloomFilter bloomFilter(colName: String, expectedNumItems: Long, numBits: Long): BloomFilter</pre>
corr	<pre>corr(col1: String, col2: String): Double corr(col1: String, col2: String, method: String): Double</pre>
countMinSketch	<pre>countMinSketch(col: Column, eps: Double, confidence: Double, seed: Int): CountMinSketch countMinSketch(col: Column, depth: Int, width: Int, seed: Int): CountMinSketch countMinSketch(colName: String, eps: Double, confidence: Double, seed: Int): CountMinSketch countMinSketch(colName: String, depth: Int, width: Int, seed: Int): CountMinSketch</pre>
cov	<pre>cov(col1: String, col2: String): Double</pre>
crosstab	<pre>crosstab(col1: String, col2: String): DataFrame</pre>
freqItems	<pre>freqItems(cols: Array[String]): DataFrame freqItems(cols: Array[String], support: Double): DataFrame freqItems(cols: Seq[String]): DataFrame freqItems(cols: Seq[String], support: Double): DataFrame</pre>
sampleBy	<pre>sampleBy[T](col: String, fractions: Map[T, Double], seed: Long): DataFrame</pre>

`DataFrameStatFunctions` is available using `stat` untyped transformation.

```
val q: DataFrame = ...
q.stat
```

approxQuantile Method

```
approxQuantile(
  cols: Array[String],
  probabilities: Array[Double],
  relativeError: Double): Array[Array[Double]]
approxQuantile(
  col: String,
  probabilities: Array[Double],
  relativeError: Double): Array[Double]
```

approxQuantile ...FIXME

bloomFilter Method

```
bloomFilter(col: Column, expectedNumItems: Long, fpp: Double): BloomFilter
bloomFilter(col: Column, expectedNumItems: Long, numBits: Long): BloomFilter
bloomFilter(colName: String, expectedNumItems: Long, fpp: Double): BloomFilter
bloomFilter(colName: String, expectedNumItems: Long, numBits: Long): BloomFilter
```

bloomFilter ...FIXME

buildBloomFilter Internal Method

```
buildBloomFilter(col: Column, zero: BloomFilter): BloomFilter
```

buildBloomFilter ...FIXME

Note	convertToDouble is used when...FIXME
------	--------------------------------------

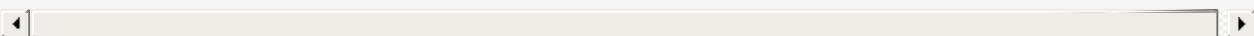
corr Method

```
corr(col1: String, col2: String): Double
corr(col1: String, col2: String, method: String): Double
```

corr ...FIXME

countMinSketch Method

```
countMinSketch(col: Column, eps: Double, confidence: Double, seed: Int): CountMinSketch  
  
countMinSketch(col: Column, depth: Int, width: Int, seed: Int): CountMinSketch  
countMinSketch(colName: String, eps: Double, confidence: Double, seed: Int): CountMinSketch  
countMinSketch(colName: String, depth: Int, width: Int, seed: Int): CountMinSketch  
// PRIVATE API  
countMinSketch(col: Column, zero: CountMinSketch): CountMinSketch
```



```
countMinSketch ...FIXME
```

cov Method

```
cov(col1: String, col2: String): Double
```

```
cov ...FIXME
```

crosstab Method

```
crosstab(col1: String, col2: String): DataFrame
```

```
crosstab ...FIXME
```

freqItems Method

```
freqItems(cols: Array[String]): DataFrame  
freqItems(cols: Array[String], support: Double): DataFrame  
freqItems(cols: Seq[String]): DataFrame  
freqItems(cols: Seq[String], support: Double): DataFrame
```

```
freqItems ...FIXME
```

sampleBy Method

```
sampleBy[T](col: String, fractions: Map[T, Double], seed: Long): DataFrame
```

```
sampleBy ...FIXME
```


Column

`Column` represents a column in a `Dataset` that holds a Catalyst `Expression` that produces a value per row.

Note	A <code>Column</code> is a value generator for every row in a <code>Dataset</code> .
------	--

A special column `*` references all columns in a `Dataset`.

With the `implicits` conversions imported, you can create "free" column references using Scala's symbols.

```
val spark: SparkSession = ...
import spark.implicits._

import org.apache.spark.sql.Column
scala> val nameCol: Column = 'name
nameCol: org.apache.spark.sql.Column = name
```

Note	<i>"Free" column references</i> are <code>Column</code> s with no association to a <code>Dataset</code> .
------	---

You can also create free column references from `$`-prefixed strings.

```
// Note that $ alone creates a ColumnName
scala> val idCol = $"id"
idCol: org.apache.spark.sql.ColumnName = id

import org.apache.spark.sql.Column

// The target type triggers the implicit conversion to Column
scala> val idCol: Column = $"id"
idCol: org.apache.spark.sql.Column = id
```

Beside using the `implicits` conversions, you can create columns using `col` and `column` functions.

```
import org.apache.spark.sql.functions._

scala> val nameCol = col("name")
nameCol: org.apache.spark.sql.Column = name

scala> val cityCol = column("city")
cityCol: org.apache.spark.sql.Column = city
```

Finally, you can create a bound `Column` using the `Dataset` the column is supposed to be part of using `Dataset.apply` factory method or `Dataset.col` operator.

Note

You can use bound `Column` references only with the `Dataset`s they have been created from.

```
scala> val textCol = dataset.col("text")
textCol: org.apache.spark.sql.Column = text

scala> val idCol = dataset.apply("id")
idCol: org.apache.spark.sql.Column = id

scala> val idCol = dataset("id")
idCol: org.apache.spark.sql.Column = id
```

You can reference nested columns using `.` (dot).

Table 1. Column Operators

Operator	Description
<code>as</code>	Specifying type hint about the expected return value of the column
<code>name</code>	

Note

`column` has a reference to Catalyst's `Expression` it was created for using `expr` `m`

```
scala> window('time, "5 seconds").expr
res0: org.apache.spark.sql.catalyst.expressions.Expression = timewindow('time,
```

Tip

Read about typed column references in [TypedColumn Expressions](#).

Specifying Type Hint— `as` Operator

```
as[U : Encoder]: TypedColumn[Any, U]
```

`as` creates a `TypedColumn` (that gives a type hint about the expected return value of the column).

```
scala> $"id".as[Int]
res1: org.apache.spark.sql.TypedColumn[Any,Int] = id
```

name Operator

```
name(alias: String): Column
```

name ...FIXME

Note

name is used when...FIXME

Adding Column to Dataset— withColumn Method

```
withColumn(colName: String, col: Column): DataFrame
```

`withColumn` method returns a new `DataFrame` with the new column `col` with `colName` name added.

Note

`withColumn` can replace an existing `colName` column.

```
scala> val df = Seq((1, "jeden"), (2, "dwa")).toDF("number", "polish")
df: org.apache.spark.sql.DataFrame = [number: int, polish: string]

scala> df.show
+----+-----+
|number|polish|
+----+-----+
|     1| jeden|
|     2|  dwa|
+----+-----+

scala> df.withColumn("polish", lit(1)).show
+----+-----+
|number|polish|
+----+-----+
|     1|      1|
|     2|      1|
+----+-----+
```

You can add new columns do a `dataset` using [withColumn](#) method.

```

val spark: SparkSession = ...
val dataset = spark.range(5)

// Add a new column called "group"
scala> dataset.withColumn("group", 'id % 2).show
+---+----+
| id|group|
+---+----+
|  0|    0|
|  1|    1|
|  2|    0|
|  3|    1|
|  4|    0|
+---+----+

```

Creating Column Instance For Catalyst Expression — apply Factory Method

```

val spark: SparkSession = ...
case class Word(id: Long, text: String)
val dataset = Seq(Word(0, "hello"), Word(1, "spark")).toDS

scala> val idCol = dataset.apply("id")
idCol: org.apache.spark.sql.Column = id

// or using Scala's magic a little bit
// the following is equivalent to the above explicit apply call
scala> val idCol = dataset("id")
idCol: org.apache.spark.sql.Column = id

```

like Operator

Caution	FIXME
---------	-------

```

scala> df("id") like "0"
res0: org.apache.spark.sql.Column = id LIKE 0

scala> df.filter('id like "0").show
+---+----+
| id| text|
+---+----+
|  0|hello|
+---+----+

```

Symbols As Column Names

```

scala> val df = Seq((0, "hello"), (1, "world")).toDF("id", "text")
df: org.apache.spark.sql.DataFrame = [id: int, text: string]

scala> df.select('id)
res0: org.apache.spark.sql.DataFrame = [id: int]

scala> df.select('id).show
+---+
| id|
+---+
| 0|
| 1|
+---+

```

Defining Windowing Column (Analytic Clause) — over Operator

```

over(): Column
over(window: WindowSpec): Column

```

`over` creates a **windowing column** (aka **analytic clause**) that allows to execute a **aggregate function** over a **window** (i.e. a group of records that are in *some* relation to the current record).

Tip

Read up on windowed aggregation in Spark SQL in [Window Aggregate Functions](#).

```

scala> val overUnspecifiedFrame = $"someColumn".over()
overUnspecifiedFrame: org.apache.spark.sql.Column = someColumn OVER (UnspecifiedFrame)

import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.WindowSpec
val spec: WindowSpec = Window.rangeBetween(Window.unboundedPreceding, Window.currentRow)
scala> val overRange = $"someColumn" over spec
overRange: org.apache.spark.sql.Column = someColumn OVER (RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

```

cast Operator

`cast` method casts a column to a data type. It makes for type-safe maps with [Row](#) objects of the proper type (not `Any`).

```
cast(to: String): Column
cast(to: DataType): Column
```

`cast` uses [CatalystSqlParser](#) to parse the data type from its canonical string representation.

cast Example

```
scala> val df = Seq((0f, "hello")).toDF("label", "text")
df: org.apache.spark.sql.DataFrame = [label: float, text: string]

scala> df.printSchema
root
 |-- label: float (nullable = false)
 |-- text: string (nullable = true)

// without cast
import org.apache.spark.sql.Row
scala> df.select("label").map { case Row(label) => label.getClass.getName }.show(false)
)
+-----+
|value      |
+-----+
|java.lang.Float|
+-----+

// with cast
import org.apache.spark.sql.types.DoubleType
scala> df.select(col("label").cast(DoubleType)).map { case Row(label) => label.getClass.getName }.show(false)
+-----+
|value      |
+-----+
|java.lang.Double|
+-----+
```

generateAlias Method

```
generateAlias(e: Expression): String
```

`generateAlias` ...FIXME

Note

- `generateAlias` is used when:
- `Column` is requested to `named`
 - `RelationalGroupedDataset` is requested to `alias`

named Method

`named: NamedExpression`

`named ...FIXME`

Note

- `named` is used when the following operators are used:
- `Dataset.select`
 - `KeyValueGroupedDataset.agg`

Column API—Column Operators

Column API is a [set of operators](#) to work with values in a column (of a Dataset).

Table 1. Column Operators

Operator	Description
asc	asc: Column
asc_nulls_first	asc_nulls_first: Column
asc_nulls_last	asc_nulls_last: Column
desc	desc: Column
desc_nulls_first	desc_nulls_first: Column
desc_nulls_last	desc_nulls_last: Column
isin	isin(list: Any*): Column
isInCollection	<p>isin(list: Any*): Column</p> <p>(New in 2.4.4) An expression operator that is <code>true</code> if the value of the column is in the given <code>values</code> collection</p> <p><code>isInCollection</code> is simply a synonym of isin operator.</p>

isin Operator

```
isin(list: Any*): Column
```

Internally, `isin` creates a `Column` with [In](#) predicate expression.

```
val ids = Seq((1, 2, 2), (2, 3, 1)).toDF("x", "y", "id")
scala> ids.show
+---+---+---+
| x| y| id|
+---+---+---+
| 1| 2| 2|
| 2| 3| 1|
+---+---+---+

val c = $"id" isin($"x", $y)
val q = ids.filter(c)
scala> q.show
+---+---+---+
| x| y| id|
+---+---+---+
| 1| 2| 2|
+---+---+---+

// Note that isin accepts non-Column values
val c = $"id" isin ("x", "y")
val q = ids.filter(c)
scala> q.show
+---+---+---+
| x| y| id|
+---+---+---+
+---+---+---+
```

TypedColumn

`TypedColumn` is a [Column](#) with the [ExpressionEncoder](#) for the types of the input and the output.

`TypedColumn` is created using [as](#) operator on a `Column`.

```
scala> val id = $"id".as[Int]
id: org.apache.spark.sql.TypedColumn[Any, Int] = id

scala> id.expr
res1: org.apache.spark.sql.catalyst.expressions.Expression = 'id
```

name Operator

```
name(alias: String): TypedColumn[T, U]
```

Note	<code>name</code> is part of Column Contract to...FIXME.
------	--

`name` ...FIXME

Note	<code>name</code> is used when...FIXME
------	--

Creating TypedColumn — `withInputType` Internal Method

```
withInputType(
  inputEncoder: ExpressionEncoder[_],
  inputAttributes: Seq[Attribute]): TypedColumn[T, U]
```

`withInputType` ...FIXME

Note	<code>withInputType</code> is used when the following typed operators are used: <ul style="list-style-type: none"> • Dataset.select • KeyValueGroupedDataset.agg • RelationalGroupedDataset.agg
------	--

Creating TypedColumn Instance

`TypedColumn` takes the following when created:

- Catalyst [expression](#)
- [ExpressionEncoder](#) of the column results

`TypedColumn` initializes the [internal registries and counters](#).

Basic Aggregation — Typed and Untyped Grouping Operators

You can calculate aggregates over a group of rows in a [Dataset](#) using [aggregate operators](#) (possibly with [aggregate functions](#)).

Table 1. Aggregate Operators

Operator	Return Type	Description
agg	RelationalGroupedDataset	Aggregates with or without grouping (i.e. over an entire Dataset)
groupBy	RelationalGroupedDataset	Used for untyped aggregates using DataFrames. Grouping is described using column expressions or column names.
groupByKey	KeyValueGroupedDataset	Used for typed aggregates using Datasets with records grouped by a key-defining discriminator function.

Note	Aggregate functions without aggregate operators return a single value. If you want to find the aggregate values for each unique value (in a column), you should groupBy first (over this column) to build the groups.
Note	<p>You can also use SparkSession to execute <i>good ol'</i> SQL with <code>GROUP BY</code> should you prefer.</p> <pre>val spark: SparkSession = ??? spark.sql("SELECT COUNT(*) FROM sales GROUP BY city")</pre> <p>SQL or Dataset API's operators go through the same query planning and optimizations, and have the same performance characteristic in the end.</p>

Aggregates Over Subset Of or Whole Dataset — `agg` Operator

```
agg(expr: Column, exprs: Column*): DataFrame
agg(exprs: Map[String, String]): DataFrame
agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame
```

`agg` applies an aggregate function on a subset or the entire `Dataset` (i.e. considering the entire data set as one group).

Note

`agg` on a `Dataset` is simply a shortcut for `groupBy().agg(...)`.

```
scala> spark.range(10).agg(sum('id) as "sum").show
+---+
|sum|
+---+
| 45|
+---+
```

`agg` can compute aggregate expressions on all the records in a `Dataset`.

Untyped Grouping — `groupBy` Operator

```
groupBy(cols: Column*): RelationalGroupedDataset
groupBy(col1: String, cols: String*): RelationalGroupedDataset
```

`groupBy` operator groups the rows in a `Dataset` by columns (as [Column expressions or names](#)).

`groupBy` gives a [RelationalGroupedDataset](#) to execute aggregate functions or operators.

```
// 10^3-record large data set
val ints = 1 to math.pow(10, 3).toInt
val nms = ints.toDF("n").withColumn("m", 'n % 2)
scala> nms.count
res0: Long = 1000

val q = nms.
  groupBy('m).
  agg(sum('n) as "sum").
  orderBy('m)
scala> q.show
+---+-----+
| m| sum|
+---+-----+
| 0|250500|
| 1|250000|
+---+-----+
```

Internally, `groupBy` [resolves column names](#) (possibly quoted) and [creates](#) a `RelationalGroupedDataset` (with `groupType` being `GroupByType`).

Note	The following uses the data setup as described in Test Setup section below.
------	---

```
scala> tokens.show
+---+-----+-----+
|name|productId|score|
+---+-----+-----+
| aaa|     100| 0.12|
| aaa|     200| 0.29|
| bbb|     200| 0.53|
| bbb|     300| 0.42|
+---+-----+-----+

scala> tokens.groupBy('name).avg().show
+---+-----+-----+
|name|avg(productId)|avg(score)|
+---+-----+-----+
| aaa|      150.0|    0.205|
| bbb|      250.0|    0.475|
+---+-----+-----+

scala> tokens.groupBy('name, 'productId).agg(Map("score" -> "avg")).show
+---+-----+-----+
|name|productId|avg(score)|
+---+-----+-----+
| aaa|     200| 0.29|
| bbb|     200| 0.53|
| bbb|     300| 0.42|
| aaa|     100| 0.12|
+---+-----+-----+

scala> tokens.groupBy('name).count.show
+---+-----+
|name|count|
+---+-----+
| aaa| 2|
| bbb| 2|
+---+-----+

scala> tokens.groupBy('name).max("score").show
+---+-----+
|name|max(score)|
+---+-----+
| aaa| 0.29|
| bbb| 0.53|
+---+-----+

scala> tokens.groupBy('name).sum("score").show
+---+-----+
|name|sum(score)|
+---+-----+
| aaa| 0.41|
| bbb| 0.95|
```

```
+----+-----+
scala> tokens.groupBy('productId).sum("score").show
+-----+-----+
|productId|      sum(score)|
+-----+-----+
|     300|       0.42|
|     100|       0.12|
|     200| 0.8200000000000001|
+-----+-----+
```

Typed Grouping — `groupByKey` Operator

```
groupByKey[K: Encoder](func: T => K): KeyValueGroupedDataset[K, T]
```

`groupByKey` groups records (of type `T`) by the input `func` and in the end returns a `KeyValueGroupedDataset` to apply aggregation to.

Note	<code>groupByKey</code> is <code>Dataset</code> 's experimental API.
------	--

```
+----+-----+
|value|count(1)|
+----+-----+
| 100|     1|
| 200|     2|
| 300|     1|
+----+-----+

import org.apache.spark.sql.expressions.scalalang._
val q = tokens.
  groupByKey(_.productId).
  agg(typed.sum[Token](_.score)).
  toDF("productId", "sum").
  orderBy('productId)
scala> q.show
+-----+-----+
|productId|      sum|
+-----+-----+
|     100|       0.12|
|     200| 0.8200000000000001|
|     300|       0.42|
+-----+
```

Test Setup

This is a setup for learning `GroupedData`. Paste it into Spark Shell using `:paste`.

```
import spark.implicits._

case class Token(name: String, productId: Int, score: Double)
val data = Seq(
  Token("aaa", 100, 0.12),
  Token("aaa", 200, 0.29),
  Token("bbb", 200, 0.53),
  Token("bbb", 300, 0.42))
val tokens = data.toDS.cache (1)
```

1. Cache the dataset so the following queries won't load/recompute data over and over again.

RelationalGroupedDataset — Untyped Row-based Grouping

`RelationalGroupedDataset` is an interface to calculate aggregates over groups of rows in a `DataFrame`.

Note	<code>KeyValueGroupedDataset</code> is used for typed aggregates over groups of custom Scala objects (not <code>Rows</code>).
------	--

`RelationalGroupedDataset` is a result of executing the following grouping operators:

- [groupBy](#)
- [rollup](#)
- [cube](#)
- [pivot](#)

Table 1. RelationalGroupedDataset's Aggregate Operators

Operator	Description
agg	
avg	
count	
max	
mean	
min	
pivot	<pre>pivot(pivotColumn: String): RelationalGroupedDataset pivot(pivotColumn: String, values: Seq[Any]): RelationalGroupedDataset pivot(pivotColumn: Column): RelationalGroupedDataset (1) pivot(pivotColumn: Column, values: Seq[Any]): RelationalGroupedDataset (1)</pre> <p>1. New in 2.4.0</p> <p>Pivots on a column (with new columns per distinct value)</p>
sum	

Note

`spark.sql.retainGroupColumns` configuration property controls whether to retain columns used for aggregation or not (in `RelationalGroupedDataset` operators).

`spark.sql.retainGroupColumns` is enabled by default.

```
scala> spark.conf.get("spark.sql.retainGroupColumns")
res1: String = true

// Use dataFrameRetainGroupColumns method for type-safe access to the current configuration
import spark.sessionState.conf
scala> conf.dataFrameRetainGroupColumns
res2: Boolean = true
```

Computing Aggregates Using Aggregate Column Expressions or Function Names — `agg` Operator

```
agg(expr: Column, exprs: Column*): DataFrame
agg(exprs: Map[String, String]): DataFrame
agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame
```

`agg` creates a `DataFrame` with the rows being the result of executing grouping expressions (specified using `columns` or names) over row groups.

Note

You can use `untyped` or `typed` column expressions.

```
val countsAndSums = spark.
  range(10). // <-- 10-element Dataset
  withColumn("group", 'id % 2). // <-- define grouping column
  groupBy("group"). // <-- group by groups
  agg(count("id") as "count", sum("id") as "sum")
scala> countsAndSums.show
+-----+-----+
|group|count|sum|
+-----+-----+
|    0|     5|  20|
|    1|     5|  25|
+-----+-----+
```

Internally, `agg` creates a `DataFrame` with `Aggregate` or `Pivot` logical operators.

```
// groupBy above
scala> println(countsAndSums.queryExecution.logical.numberedTreeString)
00 'Aggregate [group#179L], [group#179L, count('id) AS count#188, sum('id) AS sum#190]
01 +- Project [id#176L, (id#176L % cast(2 as bigint)) AS group#179L]
02   +- Range (0, 10, step=1, splits=Some(8))

// rollup operator
val rollupQ = spark.range(2).rollup('id).agg(count('id))
scala> println(rollupQ.queryExecution.logical.numberedTreeString)
00 'Aggregate [rollup('id)], [unresolvedalias('id, None), count('id) AS count(id)#267]
01 +- Range (0, 2, step=1, splits=Some(8))

// cube operator
val cubeQ = spark.range(2).cube('id).agg(count('id))
scala> println(cubeQ.queryExecution.logical.numberedTreeString)
00 'Aggregate [cube('id)], [unresolvedalias('id, None), count('id) AS count(id)#280]
01 +- Range (0, 2, step=1, splits=Some(8))

// pivot operator
val pivotQ = spark.
  range(10).
  withColumn("group", 'id % 2).
  groupBy("group").
  pivot("group").
  agg(count("id"))
scala> println(pivotQ.queryExecution.logical.numberedTreeString)
00 'Pivot [group#296L], group#296: bigint, [0, 1], [count('id)]
01 +- Project [id#293L, (id#293L % cast(2 as bigint)) AS group#296L]
02   +- Range (0, 10, step=1, splits=Some(8))
```

Creating DataFrame from Aggregate Expressions — `toDF` Internal Method

```
toDF(aggExprs: Seq[Expression]): DataFrame
```

Caution	FIXME
---------	-------

Internally, `toDF` branches off per group type.

Caution	FIXME
---------	-------

For `PivotType`, `toDF` creates a DataFrame with `Pivot` unary logical operator.

Note

`toDF` is used when the following `RelationalGroupedDataset` operators are used:

- `agg` and `count`
- `mean`, `max`, `avg`, `min` and `sum` (indirectly through `aggregateNumericColumns`)

aggregateNumericColumns Internal Method

```
aggregateNumericColumns(colNames: String*)(f: Expression => AggregateFunction): DataFrame
```

`aggregateNumericColumns ...FIXME`

Note

`aggregateNumericColumns` is used when the following `RelationalGroupedDataset` operators are used: `mean`, `max`, `avg`, `min` and `sum`.

Creating RelationalGroupedDataset Instance

`RelationalGroupedDataset` takes the following when created:

- `DataFrame`
- Grouping `expressions`
- Group type (to indicate the "source" operator)
 - `GroupByType` for `groupBy`
 - `CubeType`
 - `RollupType`
 - `PivotType`

pivot Operator

```
pivot(pivotColumn: String): RelationalGroupedDataset (1)
pivot(pivotColumn: String, values: Seq[Any]): RelationalGroupedDataset (2)
pivot(pivotColumn: Column): RelationalGroupedDataset (3)
pivot(pivotColumn: Column, values: Seq[Any]): RelationalGroupedDataset (3)
```

1. Selects distinct and sorted values on `pivotColumn` and calls the other `pivot` (that results in 3 extra "scanning" jobs)
2. Preferred as more efficient because the unique values are already provided
3. **New in 2.4.0**

`pivot` pivots on a `pivotColumn` column, i.e. adds new columns per distinct values in `pivotColumn`.

Note	<code>pivot</code> is only supported after <code>groupBy</code> operation.
------	--

Note	Only one <code>pivot</code> operation is supported on a <code>RelationalGroupedDataset</code> .
------	---

```

val visits = Seq(
  (0, "Warsaw", 2015),
  (1, "Warsaw", 2016),
  (2, "Boston", 2017)
).toDF("id", "city", "year")

val q = visits
  .groupBy("city") // <-- rows in pivot table
  .pivot("year")   // <-- columns (unique values queried)
  .count()         // <-- values in cells
scala> q.show
+-----+-----+-----+
| city|2015|2016|2017|
+-----+-----+-----+
|Warsaw|    1|    1| null|
|Boston| null| null|    1|
+-----+-----+-----+

scala> q.explain
== Physical Plan ==
HashAggregate(keys=[city#8], functions=[pivotfirst(year#9, count(1) AS `count`#222L, 2015, 2016, 2017, 0, 0)])
+- Exchange hashpartitioning(city#8, 200)
  +- HashAggregate(keys=[city#8], functions=[partial_pivotfirst(year#9, count(1) AS `count`#222L, 2015, 2016, 2017, 0, 0)])
    +- *HashAggregate(keys=[city#8, year#9], functions=[count(1)])
      +- Exchange hashpartitioning(city#8, year#9, 200)
        +- *HashAggregate(keys=[city#8, year#9], functions=[partial_count(1)])
          +- LocalTableScan [city#8, year#9]

scala> visits
  .groupBy('city)
  .pivot("year", Seq("2015")) // <-- one column in pivot table
  .count
  .show
+-----+
| city|2015|
+-----+
|Warsaw|    1|
|Boston| null|
+-----+

```

Important Use `pivot` with a list of distinct values to pivot on so Spark does not have to compute the list itself (and run three extra "scanning" jobs).

Completed Jobs (5)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	show at <console>:28	2017/04/19 09:58:23	74 ms	1/1 (2 skipped)	75/75 (203 skipped)
3	show at <console>:28	2017/04/19 09:58:23	0.1 s	1/1 (2 skipped)	100/100 (203 skipped)
2	show at <console>:28	2017/04/19 09:58:23	22 ms	1/1 (2 skipped)	20/20 (203 skipped)
1	show at <console>:28	2017/04/19 09:58:23	11 ms	1/1 (2 skipped)	4/4 (203 skipped)
0	show at <console>:28	2017/04/19 09:58:22	1 s	3/3	204/204

Figure 1. pivot in web UI (Distinct Values Defined Explicitly)

Completed Jobs (8)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	show at <console>:29	2017/04/19 09:51:20	36 ms	1/1 (2 skipped)	75/75 (203 skipped)
6	show at <console>:29	2017/04/19 09:51:20	94 ms	1/1 (2 skipped)	100/100 (203 skipped)
5	show at <console>:29	2017/04/19 09:51:20	12 ms	1/1 (2 skipped)	20/20 (203 skipped)
4	show at <console>:29	2017/04/19 09:51:20	6 ms	1/1 (2 skipped)	4/4 (203 skipped)
3	show at <console>:29	2017/04/19 09:51:19	0.6 s	3/3	204/204
2	pivot at <console>:28	2017/04/19 09:51:19	8 ms	1/1 (2 skipped)	3/3 (203 skipped)
1	pivot at <console>:28	2017/04/19 09:51:18	0.4 s	2/2 (1 skipped)	201/201 (3 skipped)
0	pivot at <console>:28	2017/04/19 09:51:17	0.8 s	2/2	203/203

Scanning jobs

Figure 2. pivot in web UI — Three Extra Scanning Jobs Due to Unspecified Distinct Values

Note	<code>spark.sql.pivotMaxValues</code> (default: 10000) controls the maximum number of (distinct) values that will be collected without error (when doing <code>pivot</code> without specifying the values for the pivot column).
------	--

Internally, `pivot` creates a `RelationalGroupedDataset` with `PivotType` group type and `pivotColumn` resolved using the `DataFrame`'s columns with `values as Literal` expressions.

	<code>toDF</code> internal method maps <code>PivotType</code> group type to a <code>DataFrame</code> with <code>Pivot</code> unary logical operator.
Note	<pre>scala> q.queryExecution.logical res0: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan = Pivot [city#8], year#9: int, [2015, 2016, 2017], [count(1) AS count#24L] +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9] +- LocalRelation [_1#3, _2#4, _3#5]</pre>

strToExpr Internal Method

```
strToExpr(expr: String): (Expression => Expression)
```

`strToExpr ...FIXME`

Note	<code>strToExpr</code> is used exclusively when <code>RelationalGroupedDataset</code> is requested to <code>agg</code> with aggregation functions specified by name
------	---

alias Method

```
alias(expr: Expression): NamedExpression
```

`alias ...FIXME`

Note	<code>alias</code> is used exclusively when <code>RelationalGroupedDataset</code> is requested to create a <code>DataFrame</code> from aggregate expressions.
------	---

KeyValueGroupedDataset — Typed Grouping

`KeyValueGroupedDataset` is an experimental interface to calculate aggregates over groups of objects in a typed Dataset.

Note	<code>RelationalGroupedDataset</code> is used for untyped Row-based aggregates.
------	---

`KeyValueGroupedDataset` is created using `Dataset.groupByKey` operator.

```
val dataset: Dataset[Token] = ...
scala> val tokensByName = dataset.groupByKey(_.name)
tokensByName: org.apache.spark.sql.KeyValueGroupedDataset[String,Token] = org.apache.spark.sql.KeyValueGroupedDataset@1e3aad46
```

Table 1. KeyValueGroupedDataset's Aggregate Operators (KeyValueGroupedDataset API)

Operator	Description
<code>agg</code>	
<code>cogroup</code>	
<code>count</code>	
<code>flatMapGroups</code>	
<code>flatMapGroupsWithState</code>	
<code>keys</code>	
<code>keyAs</code>	
<code>mapGroups</code>	
<code>mapGroupsWithState</code>	
<code>mapValues</code>	
<code>reduceGroups</code>	

`KeyValueGroupedDataset` holds keys that were used for the object.

```
scala> tokensByName.keys.show
+----+
|value|
+----+
| aaa|
| bbb|
+----+
```

aggUntyped Internal Method

```
aggUntyped(columns: TypedColumn[_, _]*): Dataset[_]
```

aggUntyped ...FIXME

Note	aggUntyped is used exclusively when KeyValueGroupedDataset.agg typed operator is used.
------	--

logicalPlan Internal Method

```
logicalPlan: AnalysisBarrier
```

logicalPlan ...FIXME

Note	logicalPlan is used when...FIXME
------	----------------------------------

Dataset Join Operators

From PostgreSQL's [2.6. Joins Between Tables](#):

Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a **join query**.

You can join two datasets using the [join operators](#) with an optional [join condition](#).

Table 1. Join Operators

Operator	Return Type	Description
<code>crossJoin</code>	<code>DataFrame</code>	Untyped <code>Row</code> -based cross join
<code>join</code>	<code>DataFrame</code>	Untyped <code>Row</code> -based join
<code>joinWith</code>	<code>Dataset</code>	Used for a type-preserving join with two output columns for records for which a join condition holds

You can also use [SQL mode](#) to join datasets using *good ol' SQL*.

```
val spark: SparkSession = ...
spark.sql("select * from t1, t2 where t1.id = t2.id")
```

You can specify a **join condition** (aka *join expression*) as part of join operators or using [where](#) or [filter](#) operators.

```
df1.join(df2, $"df1Key" === $"df2Key")
df1.join(df2).where($"df1Key" === $"df2Key")
df1.join(df2).filter($"df1Key" === $"df2Key")
```

You can specify the **join type** as part of join operators (using `joinType` optional parameter).

```
df1.join(df2, $"df1Key" === $"df2Key", "inner")
```

Table 2. Join Types

SQL	Name (joinType)	JoinType
CROSS	cross	Cross
INNER	inner	Inner
FULL OUTER	outer , full , fullouter	FullOuter
LEFT ANTI	leftanti	LeftAnti
LEFT OUTER	leftouter , left	LeftOuter
LEFT SEMI	leftsemi	LeftSemi
RIGHT OUTER	rightouter , right	RightOuter
NATURAL	Special case for Inner , LeftOuter , RightOuter , FullOuter	NaturalJoin
USING	Special case for Inner , LeftOuter , LeftSemi , RightOuter , FullOuter , LeftAnti	UsingJoin

`ExistenceJoin` is an artifical join type used to express an existential sub-query, that is often referred to as **existential join**.

Note

`LeftAnti` and `ExistenceJoin` are special cases of `LeftOuter`.

You can also find that Spark SQL uses the following two families of joins:

- `InnerLike` with `Inner` and `Cross`
- `LeftExistence` with `LeftSemi`, `LeftAnti` and `ExistenceJoin`

Tip

Name are case-insensitive and can use the underscore (`_`) at any position, i.e. `left_anti` and `LEFT_ANTI` are equivalent.

Note

Spark SQL offers different [join strategies](#) with [Broadcast Joins \(aka Map-Side Joins\)](#) among them that are supposed to optimize your join queries over large distributed datasets.

join Operators

```
join(right: Dataset[_]): DataFrame (1)
join(right: Dataset[_], usingColumn: String): DataFrame (2)
join(right: Dataset[_], usingColumns: Seq[String]): DataFrame (3)
join(right: Dataset[_], usingColumns: Seq[String], joinType: String): DataFrame (4)
join(right: Dataset[_], joinExprs: Column): DataFrame (5)
join(right: Dataset[_], joinExprs: Column, joinType: String): DataFrame (6)
```

1. Condition-less inner join
2. Inner join with a single column that exists on both sides
3. Inner join with columns that exist on both sides
4. Equi-join with explicit join type
5. Inner join
6. Join with explicit join type. Self-joins are acceptable.

`join` joins two `Dataset`s.

```
val left = Seq((0, "zero"), (1, "one")).toDF("id", "left")
val right = Seq((0, "zero"), (2, "two"), (3, "three")).toDF("id", "right")

// Inner join
scala> left.join(right, "id").show
+---+---+
| id|left|right|
+---+---+
|  0|zero| zero|
+---+---+

scala> left.join(right, "id").explain
== Physical Plan ==
*Project [id#50, left#51, right#61]
+- *BroadcastHashJoin [id#50], [id#60], Inner, BuildRight
  :- LocalTableScan [id#50, left#51]
  +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as
bigint)))
    +- LocalTableScan [id#60, right#61]

// Full outer
scala> left.join(right, Seq("id"), "fullouter").show
+---+---+
| id|left|right|
+---+---+
|  1| one| null|
|  3|null|three|
|  2|null|  two|
|  0|zero| zero|
+---+---+
```

```

scala> left.join(right, Seq("id"), "fullouter").explain
== Physical Plan ==
*Project [coalesce(id#50, id#60) AS id#85, left#51, right#61]
+- SortMergeJoin [id#50], [id#60], FullOuter
  :- *Sort [id#50 ASC NULLS FIRST], false, 0
  :  +- Exchange hashpartitioning(id#50, 200)
  :    +- LocalTableScan [id#50, left#51]
  +- *Sort [id#60 ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(id#60, 200)
      +- LocalTableScan [id#60, right#61]

// Left anti
scala> left.join(right, Seq("id"), "leftanti").show
+---+---+
| id|left|
+---+---+
|  1| one|
+---+---+

scala> left.join(right, Seq("id"), "leftanti").explain
== Physical Plan ==
*BroadcastHashJoin [id#50], [id#60], LeftAnti, BuildRight
:- LocalTableScan [id#50, left#51]
+- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as big
int)))
  +- LocalTableScan [id#60]

```

Internally, `join(right: Dataset[_])` creates a DataFrame with a condition-less Join logical operator (in the current SparkSession).

Note	<code>join(right: Dataset[_])</code> creates a logical plan with a condition-less Join operator with two child logical plans of the both sides of the join.
------	---

Note	<code>join(right: Dataset[_], usingColumns: Seq[String], joinType: String)</code> creates a logical plan with a condition-less Join operator with UsingJoin join type.
------	--

	<code>join(right: Dataset[_], joinExprs: Column, joinType: String)</code> accepts self-joins where <code>joinExprs</code> is of the form:
--	---

```
df("key") === df("key")
```

Note	<p>That is usually considered a trivially true condition and refused as acceptable.</p> <p>With <code>spark.sql.selfJoinAutoResolveAmbiguity</code> option enabled (which it is by default), <code>join</code> will automatically resolve ambiguous join conditions into ones that might make sense.</p>
------	--

See [\[SPARK-6231\] Join on two tables \(generated from same one\) is broken.](#)

crossJoin Method

```
crossJoin(right: Dataset[_]): DataFrame
```

`crossJoin` joins two [Datasets](#) using [Cross](#) join type with no condition.

Note

`crossJoin` creates an explicit cartesian join that can be very expensive without an extra filter (that can be pushed down).

Type-Preserving Joins — `joinWith` Operators

```
joinWith[U](other: Dataset[U], condition: Column): Dataset[(T, U)] (1)
joinWith[U](other: Dataset[U], condition: Column, joinType: String): Dataset[(T, U)]
```

1. inner equi-join

`joinWith` creates a [Dataset](#) with two columns `_1` and `_2` that each contain records for which `condition` holds.

```

case class Person(id: Long, name: String, cityId: Long)
case class City(id: Long, name: String)
val family = Seq(
  Person(0, "Agata", 0),
  Person(1, "Iweta", 0),
  Person(2, "Patryk", 2),
  Person(3, "Maksym", 0)).toDS
val cities = Seq(
  City(0, "Warsaw"),
  City(1, "Washington"),
  City(2, "Sopot")).toDS

val joined = family.joinWith(cities, family("cityId") === cities("id"))
scala> joined.printSchema
root
|-- _1: struct (nullable = false)
|   |-- id: long (nullable = false)
|   |-- name: string (nullable = true)
|   |-- cityId: long (nullable = false)
|-- _2: struct (nullable = false)
|   |-- id: long (nullable = false)
|   |-- name: string (nullable = true)
scala> joined.show
+-----+-----+
|      _1|      _2|
+-----+-----+
| [0,Agata,0]| [0,Warsaw]|
| [1,Iweta,0]| [0,Warsaw]|
| [2,Patryk,2]| [2,Sopot]|
| [3,Maksym,0]| [0,Warsaw]|
+-----+-----+

```

Note	<code>joinWith</code> preserves type-safety with the original object types.
------	---

Note	<code>joinWith</code> creates a <code>Dataset</code> with <code>Join</code> logical plan.
------	---

Broadcast Joins (aka Map-Side Joins)

Spark SQL uses **broadcast join** (aka **broadcast hash join**) instead of hash join to optimize join queries when the size of one side data is below `spark.sql.autoBroadcastJoinThreshold`.

Broadcast join can be very efficient for joins between a large table (fact) with relatively small tables (dimensions) that could then be used to perform a **star-schema join**. It can avoid sending all data of the large table over the network.

You can use `broadcast` function or SQL's `broadcast hints` to mark a dataset to be broadcast when used in a join query.

Note	According to the article Map-Side Join in Spark , broadcast join is also called a replicated join (in the distributed system community) or a map-side join (in the Hadoop community).
------	--

`CanBroadcast` object matches a [LogicalPlan](#) with output small enough for broadcast join.

Note	Currently statistics are only supported for Hive Metastore tables where the command <code>ANALYZE TABLE [tableName] COMPUTE STATISTICS noscan</code> has been run.
------	--

[JoinSelection](#) execution planning strategy uses `spark.sql.autoBroadcastJoinThreshold` property (default: `10M`) to control the size of a dataset before broadcasting it to all worker nodes when performing a join.

```

val threshold = spark.conf.get("spark.sql.autoBroadcastJoinThreshold").toInt
scala> threshold / 1024 / 1024
res0: Int = 10

val q = spark.range(100).as("a").join(spark.range(100).as("b")).where($"a.id" === $"b.id")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'Filter ('a.id = 'b.id)
01 +- Join Inner
02   :- SubqueryAlias a
03   :  +- Range (0, 100, step=1, splits=Some(8))
04   +- SubqueryAlias b
05     +- Range (0, 100, step=1, splits=Some(8))

scala> println(q.queryExecution.sparkPlan.numberedTreeString)
00 BroadcastHashJoin [id#0L], [id#4L], Inner, BuildRight
01 :- Range (0, 100, step=1, splits=8)
02 +- Range (0, 100, step=1, splits=8)

scala> q.explain
== Physical Plan ==
*BroadcastHashJoin [id#0L], [id#4L], Inner, BuildRight

```

```

:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 100, step=1, splits=8)

spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold")
res1: String = -1

scala> q.explain
== Physical Plan ==
*SortMergeJoin [id#0L], [id#4L], Inner
:- *Sort [id#0L ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(id#0L, 200)
:    +- *Range (0, 100, step=1, splits=8)
+- *Sort [id#4L ASC NULLS FIRST], false, 0
  +- ReusedExchange [id#4L], Exchange hashpartitioning(id#0L, 200)

// Force BroadcastHashJoin with broadcast hint (as function)
val qBroadcast = spark.range(100).as("a").join(broadcast(spark.range(100)).as("b")).where($"a.id" === $"b.id")
scala> qBroadcast.explain
== Physical Plan ==
*BroadcastHashJoin [id#14L], [id#18L], Inner, BuildRight
:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 100, step=1, splits=8)

// Force BroadcastHashJoin using SQL's BROADCAST hint
// Supported hints: BROADCAST, BROADCASTJOIN or MAPJOIN
val qBroadcastLeft = """
SELECT /*+ BROADCAST (lf) */ *
FROM range(100) lf, range(1000) rt
WHERE lf.id = rt.id
"""

scala> sql(qBroadcastLeft).explain
== Physical Plan ==
*BroadcastHashJoin [id#34L], [id#35L], Inner, BuildRight
:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 1000, step=1, splits=8)

val qBroadcastRight = """
SELECT /*+ MAPJOIN (rt) */ *
FROM range(100) lf, range(1000) rt
WHERE lf.id = rt.id
"""

scala> sql(qBroadcastRight).explain
== Physical Plan ==
*BroadcastHashJoin [id#42L], [id#43L], Inner, BuildRight
:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 1000, step=1, splits=8)

```


Window Aggregation

Window Aggregation is...FIXME

From Structured Query to Physical Plan

Spark Analyzer uses [ExtractWindowExpressions](#) logical resolution rule to replace (extract) [WindowExpression](#) expressions with [Window](#) logical operators in a [logical query plan](#).

Note	Window —> (BasicOperators) —> WindowExec —> WindowExec.adoc#doExecute (and windowExecBufferInMemoryThreshold + windowExecBufferSpillThreshold)
------	--

WindowSpec — Window Specification

`WindowSpec` is a **window specification** that defines which rows are included in a **window (frame)**, i.e. the set of rows that are associated with the current row by some *relation*.

`WindowSpec` takes the following when created:

- **Partition specification** (`Seq[Expression]`) which defines which records are in the same partition. With no partition defined, all records belong to a single partition
- **Ordering Specification** (`Seq[SortOrder]`) which defines how records in a partition are ordered that in turn defines the position of a record in a partition. The ordering could be ascending (`ASC` in SQL or `asc` in Scala) or descending (`DESC` or `desc`).
- **Frame Specification** (`WindowFrame`) which defines the rows to be included in the frame for the current row, based on their relative position to the current row. For example, "*the three rows preceding the current row to the current row*" describes a frame including the current input row and three rows appearing before the current row.

You use [Window object](#) to create a `WindowSpec`.

```
import org.apache.spark.sql.expressions.Window
scala> val byHTokens = Window.partitionBy('token startsWith "h")
byHTokens: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@574985d8
```

Once the initial version of a `WindowSpec` is created, you use the [methods](#) to further configure the window specification.

Table 1. WindowSpec API

Method	Description
orderBy	orderBy(cols: Column*): WindowSpec orderBy(colName: String, colNames: String*): WindowSpec
partitionBy	partitionBy(cols: Column*): WindowSpec partitionBy(colName: String, colNames: String*): WindowSpec
rangeBetween	rangeBetween(start: Column, end: Column): WindowSpec rangeBetween(start: Long, end: Long): WindowSpec
rowsBetween	rowsBetween(start: Long, end: Long): WindowSpec

With a window specification fully defined, you use [Column.over](#) operator that associates the `WindowSpec` with an [aggregate](#) or [window](#) function.

```
scala> :type windowSpec
org.apache.spark.sql.expressions.WindowSpec

import org.apache.spark.sql.functions.rank
val c = rank over windowSpec
```

withAggregate Internal Method

```
withAggregate(aggregate: Column): Column
```

`withAggregate` ...FIXME

Note	<code>withAggregate</code> is used exclusively when Column.over operator is used.
------	---

Window Utility Object — Defining Window Specification

`Window` utility object is a [set of static methods](#) to define a [window specification](#).

Table 1. Window API

Method	Description
<code>currentRow</code>	<pre>currentRow: Long</pre> <p>Value representing the current row that is used to define frame boundaries.</p>
<code>orderBy</code>	<pre>orderBy(cols: Column*): WindowSpec orderBy(colName: String, colNames: String*): WindowSpec</pre> <p>Creates a WindowSpec with the ordering defined.</p>
<code>partitionBy</code>	<pre>partitionBy(cols: Column*): WindowSpec partitionBy(colName: String, colNames: String*): WindowSpec</pre> <p>Creates a WindowSpec with the partitioning defined.</p>
<code>rangeBetween</code>	<pre>rangeBetween(start: Column, end: Column): WindowSpec rangeBetween(start: Long, end: Long): WindowSpec</pre> <p>Creates a WindowSpec with the frame boundaries defined, from <code>start</code> (inclusive) to <code>end</code> (inclusive). Both <code>start</code> and <code>end</code> are relative to the current row based on the actual value of the <code>ORDER BY</code> expression(s).</p>
<code>rowsBetween</code>	<pre>rowsBetween(start: Long, end: Long): WindowSpec</pre> <p>Creates a WindowSpec with the frame boundaries defined, from <code>start</code> (inclusive) to <code>end</code> (inclusive). Both <code>start</code> and <code>end</code> are positions relative to the current row based on the position of the row within the partition.</p>
<code>unboundedFollowing</code>	<pre>unboundedFollowing: Long</pre>

	Value representing the last row in a partition (equivalent to "UNBOUNDED FOLLOWING" in SQL) that is used to define frame boundaries .
unboundedPreceding	<p>unboundedPreceding: Long</p> <p>Value representing the first row in a partition (equivalent to "UNBOUNDED PRECEDING" in SQL) that is used to define frame boundaries.</p>

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions.{currentRow, lit}
val windowSpec = Window
  .partitionBy($"orderId")
  .orderBy($"time")
  .rangeBetween(currentRow, lit(1))
scala> :type windowSpec
org.apache.spark.sql.expressions.WindowSpec
```

Creating "Empty" WindowSpec — `spec` Internal Method

`spec: WindowSpec`

`spec` creates an "empty" [WindowSpec](#), i.e. with empty [partition](#) and [ordering](#) specifications, and a [UnspecifiedFrame](#).

Note	<p><code>spec</code> is used when:</p> <ul style="list-style-type: none"> • Column.over operator is used (with no <code>WindowSpec</code>) • Window utility object is requested to partitionBy, orderBy, rowsBetween and rangeBetween
------	---

Standard Functions — functions Object

`org.apache.spark.sql.functions` object defines built-in [standard functions](#) to work with (values produced by) [columns](#).

You can access the standard functions using the following `import` statement in your Scala application:

```
import org.apache.spark.sql.functions._
```

Table 1. (Subset of) Standard Functions in Spark SQL

	Name	Description
	approx_count_distinct	<code>approx_count_distinct(e: Column): Column</code> <code>approx_count_distinct(columnName: String): Column</code> <code>approx_count_distinct(e: Column, rsd: Double): Column</code> <code>approx_count_distinct(columnName: String, rsd: Double): Column</code>
	avg	<code>avg(e: Column): Column</code> <code>avg(columnName: String): Column</code>
	collect_list	<code>collect_list(e: Column): Column</code> <code>collect_list(columnName: String): Column</code>
	collect_set	<code>collect_set(e: Column): Column</code> <code>collect_set(columnName: String): Column</code>
	corr	<code>corr(column1: Column, column2: Column): Column</code> <code>corr(columnName1: String, columnName2: String): Column</code>
	count	<code>count(e: Column): Column</code> <code>count(columnName: String): TypedColumn[Long]</code>
	countDistinct	<code>countDistinct(expr: Column, exprs: Column): Column</code> <code>countDistinct(columnName: String, columnName1: String, columnName2: String): Column</code>

Aggregate functions	covar_pop	<pre>covar_pop(column1: Column, column2: Column): Column covar_pop(columnName1: String, columnName2: String): Column</pre>
	covar_samp	<pre>covar_samp(column1: Column, column2: Column): Column covar_samp(columnName1: String, columnName2: String): Column</pre>
	first	<pre>first(e: Column): Column first(e: Column, ignoreNulls: Boolean): Column first(columnName: String): Column first(columnName: String, ignoreNulls: Boolean): Column</pre>
		Returns the first value in a group. Returns the first value in the group. If the ignoreNulls flag is set to true, it ignores null values. If all values are null, the result is null.
	grouping	<pre>grouping(e: Column): Column grouping(columnName: String): Column</pre>
		Indicates whether a given column is aggregated or not.
	grouping_id	<pre>grouping_id(cols: Column*): Column grouping_id(colName: String, colNames: String*): Column</pre>
		Computes the level of grouping.
	kurtosis	<pre>kurtosis(e: Column): Column kurtosis(columnName: String): Column</pre>
	last	<pre>last(e: Column, ignoreNulls: Boolean): Column last(columnName: String, ignoreNulls: Boolean): Column last(e: Column): Column last(columnName: String): Column</pre>
	max	<pre>max(e: Column): Column max(columnName: String): Column</pre>
	mean	<pre>mean(e: Column): Column mean(columnName: String): Column</pre>
	min	<pre>min(e: Column): Column min(columnName: String): Column</pre>

<code>skewness</code>	<code>skewness(e: Column): Column</code> <code>skewness(columnName: String): Column</code>
<code>stddev</code>	<code>stddev(e: Column): Column</code> <code>stddev(columnName: String): Column</code>
<code>stddev_pop</code>	<code>stddev_pop(e: Column): Column</code> <code>stddev_pop(columnName: String): Column</code>
<code>stddev_samp</code>	<code>stddev_samp(e: Column): Column</code> <code>stddev_samp(columnName: String): Column</code>
<code>sum</code>	<code>sum(e: Column): Column</code> <code>sum(columnName: String): Column</code>
<code>sumDistinct</code>	<code>sumDistinct(e: Column): Column</code> <code>sumDistinct(columnName: String): Column</code>
<code>variance</code>	<code>variance(e: Column): Column</code> <code>variance(columnName: String): Column</code>
<code>var_pop</code>	<code>var_pop(e: Column): Column</code> <code>var_pop(columnName: String): Column</code>
<code>var_samp</code>	<code>var_samp(e: Column): Column</code> <code>var_samp(columnName: String): Column</code>
<code>array_contains</code>	<code>array_contains(column: Column, value: Any)</code>
<code>array_distinct</code>	<code>array_distinct(e: Column): Column</code> (New in 2.4.0)

array_except	array_except(e: Column): Column (New in 2.4.0)
array_intersect	array_intersect(col1: Column, col2: Column): Column (New in 2.4.0)
array_join	array_join(column: Column, delimiter: String): Column array_join(column: Column, delimiter: String): Column (New in 2.4.0)
array_max	array_max(e: Column): Column (New in 2.4.0)
array_min	array_min(e: Column): Column (New in 2.4.0)
array_position	array_position(column: Column, value: Any): Int (New in 2.4.0)
array_remove	array_remove(column: Column, element: Any): Column (New in 2.4.0)
array_repeat	array_repeat(e: Column, count: Int): Column array_repeat(left: Column, right: Column): Column (New in 2.4.0)
array_sort	array_sort(e: Column): Column (New in 2.4.0)

Collection functions	array_union	array_union(col1: Column, col2: Column): Column (New in 2.4.0)
	arrays_zip	arrays_zip(e: Column*): Column (New in 2.4.0)
	arrays_overlap	arrays_overlap(a1: Column, a2: Column): Column (New in 2.4.0)
	element_at	element_at(column: Column, value: Any): Any (New in 2.4.0)
	explode	explode(e: Column): Column
	explode_outer	explode_outer(e: Column): Column Creates a new row for each element in the array/map. If the array/map is <code>null</code> or empty then <code>null</code> .
	flatten	flatten(e: Column): Column (New in 2.4.0)
	from_json	from_json(e: Column, schema: Column): Column from_json(e: Column, schema: DataType): Column from_json(e: Column, schema: DataType, options: Map): Column from_json(e: Column, schema: String, options: Map): Column from_json(e: Column, schema: StructType): Column from_json(e: Column, schema: StructType, options: Map): Column

1. New in 2.4.0

Parses a column with a JSON string into a column of `StructType` elements with the specified schema.

map_concat	<code>map_concat(cols: Column*): Column</code> (New in 2.4.0)
map_from_entries	<code>map_from_entries(e: Column): Column</code> (New in 2.4.0)
map_keys	<code>map_keys(e: Column): Column</code>
map_values	<code>map_values(e: Column): Column</code>
poseplode	<code>poseplode(e: Column): Column</code>
poseplode_outer	<code>poseplode_outer(e: Column): Column</code>
reverse	<code>reverse(e: Column): Column</code> Returns a reversed string or an array with reversed elements. <div style="display: flex; justify-content: space-between;">NoteSupport for reversing arrays is experimental.</div>
schema_of_json	<code>schema_of_json(json: Column): Column</code> <code>schema_of_json(json: String): Column</code> (New in 2.4.0)
sequence	<code>sequence(start: Column, stop: Column): Column</code> <code>sequence(start: Column, stop: Column, step: Column): Column</code> (New in 2.4.0)
shuffle	<code>shuffle(e: Column): Column</code> (New in 2.4.0)

Date and time functions	<code>size</code>	<p><code>size(e: Column): Column</code></p> <p>Returns the size of the given array or map.</p>
	<code>slice</code>	<p><code>slice(x: Column, start: Int, length: Int)</code></p> <p>(New in 2.4.0)</p>
	<code>current_date</code>	<p><code>current_date(): Column</code></p>
	<code>current_timestamp</code>	<p><code>current_timestamp(): Column</code></p>
	<code>from_utc_timestamp</code>	<p><code>from_utc_timestamp(ts: Column, tz: String)</code> <code>from_utc_timestamp(ts: Column, tz: Column)</code></p> <p>1. New in 2.4.0</p>
	<code>months_between</code>	<p><code>months_between(end: Column, start: Column)</code> <code>months_between(end: Column, start: Column, column (1))</code></p> <p>1. New in 2.4.0</p>
	<code>to_date</code>	<p><code>to_date(e: Column): Column</code> <code>to_date(e: Column, fmt: String): Column</code></p>
	<code>to_timestamp</code>	<p><code>to_timestamp(s: Column): Column</code> <code>to_timestamp(s: Column, fmt: String): Column</code></p>
	<code>to_utc_timestamp</code>	<p><code>to_utc_timestamp(ts: Column, tz: String)</code> <code>to_utc_timestamp(ts: Column, tz: Column)</code></p> <p>1. New in 2.4.0</p>
	<code>unix_timestamp</code>	<p>Converts current or specified time to Unix timestamp.</p> <p><code>unix_timestamp(): Column</code> <code>unix_timestamp(s: Column): Column</code> <code>unix_timestamp(s: Column, p: String): Column</code></p>

	<code>window</code>	Generates tumbling time windows <pre>window(timeColumn: Column, windowDuration: String): Column window(timeColumn: Column, windowDuration: String, slideDuration: String): Column window(timeColumn: Column, windowDuration: String, slideDuration: String, startTime: String): Column</pre>
Regular functions (Non-aggregate functions)	<code>bin</code>	Converts the value of a long column to binary
	<code>array</code>	
	<code>broadcast</code>	
	<code>coalesce</code>	Gives the first non- <code>null</code> value among the columns
	<code>col</code> and <code>column</code>	Creating Columns
	<code>expr</code>	
	<code>lit</code>	
	<code>map</code>	
	<code>monotonically_increasing_id</code>	Returns monotonically increasing 64-bit integers. Intended to be monotonically increasing and unique,
	<code>struct</code>	
String functions	<code>typedLit</code>	
	<code>when</code>	
UDF functions	<code>split</code>	
	<code>upper</code>	
	<code>udf</code>	Creating UDFs
	<code>callUDF</code>	Executing an UDF by name with variable-length arguments

Window functions	cume_dist	cume_dist(): Column Computes the cumulative distribution of records per partition
	currentRow	currentRow(): Column
	dense_rank	dense_rank(): Column Computes the rank of records per window partition
	lag	lag(e: Column, offset: Int): Column lag(columnName: String, offset: Int): Column lag(columnName: String, offset: Int, defaultValue: Double)
	lead	lead(columnName: String, offset: Int): Column lead(e: Column, offset: Int): Column lead(columnName: String, offset: Int, defaultValue: Double) lead(e: Column, offset: Int, defaultValue: Double)
	ntile	ntile(n: Int): Column Computes the ntile group
	percent_rank	percent_rank(): Column Computes the rank of records per window partition
	rank	rank(): Column Computes the rank of records per window partition
	row_number	row_number(): Column Computes the sequential numbering per window partition
	unboundedFollowing	unboundedFollowing(): Column

unboundedPreceding	unboundedPreceding(): Column
--------------------	------------------------------

	Tip
--	-----

	The page gives only a brief overview of the many functions available in <code>functions</code> object and so you should read the official documentation of the <code>functions</code> object .
--	--

Executing UDF by Name and Variable-Length Column List — `callUDF` Function

callUDF(udfName: String, cols: Column*): Column

`callUDF` executes an UDF by `udfName` and variable-length list of columns.

Defining UDFs — `udf` Function

udf(f: FunctionN[...]): UserDefinedFunction

The `udf` family of functions allows you to create [user-defined functions \(UDFs\)](#) based on a user-defined function in Scala. It accepts `f` function of 0 to 10 arguments and the input and output types are automatically inferred (given the types of the respective input and output types of the function `f`).

```
import org.apache.spark.sql.functions._

val _length: String => Int = _.length
val _lengthUDF = udf(_length)

// define a dataframe
val df = sc.parallelize(0 to 3).toDF("num")

// apply the user-defined function to "num" column
scala> df.withColumn("len", _lengthUDF($"num")).show
+---+---+
|num|len|
+---+---+
| 0| 1|
| 1| 1|
| 2| 1|
| 3| 1|
+---+---+
```

Since Spark 2.0.0, there is another variant of `udf` function:

```
udf(f: AnyRef, dataType: DataType): UserDefinedFunction
```

`udf(f: AnyRef, dataType: DataType)` allows you to use a Scala closure for the function argument (as `f`) and explicitly declaring the output data type (as `dataType`).

```
// given the dataframe above

import org.apache.spark.sql.types.IntegerType
val byTwo = udf((n: Int) => n * 2, IntegerType)

scala> df.withColumn("len", byTwo($"num")).show
+---+---+
|num|len|
+---+---+
| 0| 0|
| 1| 2|
| 2| 4|
| 3| 6|
+---+---+
```

split Function

```
split(str: Column, pattern: String): Column
```

`split` function splits `str` column using `pattern`. It returns a new `Column`.

Note	<code>split</code> UDF uses java.lang.String.split(String regex, int limit) method.
------	---

```
val df = Seq((0, "hello|world"), (1, "witaj|swiecie")).toDF("num", "input")
val withSplit = df.withColumn("split", split($"input", "[|]"))

scala> withSplit.show
+---+-----+-----+
|num|      input|      split|
+---+-----+-----+
| 0| hello|world| [hello, world]|
| 1|witaj|swiecie|[witaj, swiecie]|
+---+-----+-----+
```

Note	<code>.\$ (){}^?*+\`</code> are RegEx's meta characters and are considered special.
------	---

upper Function

```
upper(e: Column): Column
```

`upper` function converts a string column into one with all letter upper. It returns a new `Column`.

Note

The following example uses two functions that accept a `Column` and return another to showcase how to chain them.

```
val df = Seq((0,1,"hello"), (2,3,"world"), (2,4, "ala")).toDF("id", "val", "name")
val withUpperReversed = df.withColumn("upper", reverse(upper($"name")))

scala> withUpperReversed.show
+---+---+-----+
| id|val| name|upper|
+---+---+-----+
|  0|  1|hello|OLLEH|
|  2|  3|world|DLROW|
|  2|  4|  ala|   ALA|
+---+---+-----+
```

Converting Long to Binary Format (in String Representation) — `bin` Function

```
bin(e: Column): Column
bin(columnName: String): Column (1)
```

1. Calls the first `bin` with `columnName` as a `Column`

`bin` converts the long value in a column to its binary format (i.e. as an unsigned integer in base 2) with no extra leading 0s.

```

scala> spark.range(5).withColumn("binary", bin('id)).show
+---+-----+
| id|binary|
+---+-----+
| 0|    0|
| 1|    1|
| 2|   10|
| 3|   11|
| 4|  100|
+---+-----+

val withBin = spark.range(5).withColumn("binary", bin('id))
scala> withBin.printSchema
root
|-- id: long (nullable = false)
|-- binary: string (nullable = false)

```

Internally, `bin` creates a [Column](#) with `Bin` unary expression.

```

scala> withBin.queryExecution.logical
res2: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'Project [*, bin('id) AS binary#14]
+- Range (0, 5, step=1, splits=Some(8))

```

Note	<code>Bin</code> unary expression uses java.lang.Long.toBinaryString for the conversion.
------	--

Note	<code>Bin</code> expression supports code generation (aka <code>CodeGen</code>). <pre> val withBin = spark.range(5).withColumn("binary", bin('id)) scala> withBin.queryExecution.debugcodegen Found 1 WholeStageCodegen subtrees. == Subtree 1 / 1 == *Project [id#19L, bin(id#19L) AS binary#22] +- *Range (0, 5, step=1, splits=Some(8)) ... /* 103 */ UTF8String project_value1 = null; /* 104 */ project_value1 = UTF8String.fromString(java.lang.Long.toBir </pre>
------	---

Standard Aggregate Functions

Table 1. Standard Aggregate Functions

Name	Description
approx_count_distinct	<pre>approx_count_distinct(e: Column): Column approx_count_distinct(columnName: String): Column approx_count_distinct(e: Column, rsd: Double): Column approx_count_distinct(columnName: String, rsd: Double) : Column</pre>
avg	<pre>avg(e: Column): Column avg(columnName: String): Column</pre>
collect_list	<pre>collect_list(e: Column): Column collect_list(columnName: String): Column</pre>
collect_set	<pre>collect_set(e: Column): Column collect_set(columnName: String): Column</pre>
corr	<pre>corr(column1: Column, column2: Column): Column corr(columnName1: String, columnName2: String): Column</pre>
count	<pre>count(e: Column): Column count(columnName: String): TypedColumn[Any, Long]</pre>
countDistinct	<pre>countDistinct(expr: Column, exprs: Column*): Column countDistinct(columnName: String, columnNames: String*): Column</pre>
covar_pop	<pre>covar_pop(column1: Column, column2: Column): Column covar_pop(columnName1: String, columnName2: String): Column</pre>

covar_samp	<pre>covar_samp(column1: Column, column2: Column): Column covar_samp(columnName1: String, columnName2: String): Column</pre>
first	<pre>first(e: Column): Column first(e: Column, ignoreNulls: Boolean): Column first(columnName: String): Column first(columnName: String, ignoreNulls: Boolean): Column</pre> <p>Returns the first value in a group. Returns the first non-null value when <code>ignoreNulls</code> flag on. If all values are null, then returns null.</p>
grouping	<pre>grouping(e: Column): Column grouping(columnName: String): Column</pre> <p>Indicates whether a given column is aggregated or not</p>
grouping_id	<pre>grouping_id(cols: Column*): Column grouping_id(colName: String, colNames: String*): Column</pre> <p>Computes the level of grouping</p>
kurtosis	<pre>kurtosis(e: Column): Column kurtosis(columnName: String): Column</pre>
last	<pre>last(e: Column, ignoreNulls: Boolean): Column last(columnName: String, ignoreNulls: Boolean): Column last(e: Column): Column last(columnName: String): Column</pre>
max	<pre>max(e: Column): Column max(columnName: String): Column</pre>
mean	<pre>mean(e: Column): Column mean(columnName: String): Column</pre>
min	<pre>min(e: Column): Column min(columnName: String): Column</pre>

skewness	skewness(e: Column): Column skewness(columnName: String): Column
stddev	stddev(e: Column): Column stddev(columnName: String): Column
stddev_pop	stddev_pop(e: Column): Column stddev_pop(columnName: String): Column
stddev_samp	stddev_samp(e: Column): Column stddev_samp(columnName: String): Column
sum	sum(e: Column): Column sum(columnName: String): Column
sumDistinct	sumDistinct(e: Column): Column sumDistinct(columnName: String): Column
variance	variance(e: Column): Column variance(columnName: String): Column
var_pop	var_pop(e: Column): Column var_pop(columnName: String): Column
var_samp	var_samp(e: Column): Column var_samp(columnName: String): Column

grouping Aggregate Function

```
grouping(e: Column): Column
grouping(columnName: String): Column (1)
```

1. Calls the first `grouping` with `columnName` as a `column`

`grouping` is an aggregate function that indicates whether a specified column is aggregated or not and:

- returns `1` if the column is in a subtotal and is `NULL`
- returns `0` if the underlying value is `NULL` or any other value

Note `grouping` can only be used with `cube`, `rollup` or `GROUPING SETS` multi-dimensional aggregate operators (and is verified when `Analyzer` does check analysis).

From [Hive's documentation about Grouping_ID function](#) (that can somehow help to understand `grouping`):

When aggregates are displayed for a column its value is `null`. This may conflict in case the column itself has some `null` values. There needs to be some way to identify `NULL` in column, which means aggregate and `NULL` in column, which means value. `GROUPING_ID` function is the solution to that.

```

val tmpWorkshops = Seq(
  ("Warsaw", 2016, 2),
  ("Toronto", 2016, 4),
  ("Toronto", 2017, 1)).toDF("city", "year", "count")

// there seems to be a bug with nulls
// and so the need for the following union
val cityNull = Seq(
  (null.asInstanceOf[String], 2016, 2)).toDF("city", "year", "count")

val workshops = tmpWorkshops union cityNull

scala> workshops.show
+---+---+---+
| city|year|count|
+---+---+---+
| Warsaw|2016| 2|
| Toronto|2016| 4|
| Toronto|2017| 1|
| null|2016| 2|
+---+---+---+

val q = workshops
  .cube("city", "year")
  .agg(grouping("city"), grouping("year")) // <-- grouping here
  .sort($"city".desc_nulls_last, $"year".desc_nulls_last)

scala> q.show
+---+---+-----+-----+
| city|year|grouping(city)|grouping(year)|
+---+---+-----+-----+
| Warsaw|2016|          0|          0|
| Warsaw|null|        0|          1|
| Toronto|2017|          0|          0|
| Toronto|2016|          0|          0|
| Toronto|null|        0|          1|
| null|2017|        1|          0|
| null|2016|        1|          0|
| null|2016|        0|          0|  <-- null is city
| null|null|      0|          1|  <-- null is city
| null|null|      1|          1|
+---+---+-----+-----+

```

Internally, `grouping` creates a [Column](#) with `Grouping` expression.

```

val q = workshops.cube("city", "year").agg(grouping("city"))
scala> println(q.queryExecution.logical)
'Aggregate [cube(city#182, year#183)], [city#182, year#183, grouping('city) AS groupin
g(city)#705]
+- Union
  :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
    :  +- LocalRelation [_1#178, _2#179, _3#180]
  +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
    +- LocalRelation [_1#192, _2#193, _3#194]

scala> println(q.queryExecution.analyzed)
Aggregate [city#724, year#725, spark_grouping_id#721], [city#724, year#725, cast((shif
tright(spark_grouping_id#721, 1) & 1) as tinyint) AS grouping(city)#720]
+- Expand [List(city#182, year#183, count#184, city#722, year#723, 0), List(city#182,
year#183, count#184, city#722, null, 1), List(city#182, year#183, count#184, null, ye
r#723, 2), List(city#182, year#183, count#184, null, null, 3)], [city#182, year#183, c
ount#184, city#724, year#725, spark_grouping_id#721]
  +- Project [city#182, year#183, count#184, city#182 AS city#722, year#183 AS year#7
23]
    +- Union
      :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
        :  +- LocalRelation [_1#178, _2#179, _3#180]
      +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
        +- LocalRelation [_1#192, _2#193, _3#194]

```

Note

grouping was added to Spark SQL in [\[SPARK-12706\]](#) support grouping/grouping_id function together group set.

grouping_id Aggregate Function

```

grouping_id(cols: Column*): Column
grouping_id(colName: String, colNames: String*): Column (1)

```

1. Calls the first `grouping_id` with `colName` and `colNames` as objects of type `Column`
`grouping_id` is an aggregate function that computes the level of grouping:

- `0` for combinations of each column
- `1` for subtotals of column 1
- `2` for subtotals of column 2
- And so on...

```

val tmpworkshops = Seq(
  ("Warsaw", 2016, 2),
  ("Toronto", 2016, 4),

```



```
scala> query.withColumn("bitmask", bin($"grouping_id()")).show
+-----+-----+-----+
| city|year|grouping_id()|bitmask|
+-----+-----+-----+
| Warsaw|2016|          0|      0|
| Warsaw|null|        1|      1|
| Toronto|2017|          0|      0|
| Toronto|2016|          0|      0|
| Toronto|null|        1|      1|
| null|2017|          2|     10|
| null|2016|          2|     10|
| null|2016|          0|      0| <- null is city
| null|null|        3|     11|
| null|null|        1|      1|
+-----+-----+-----+
```

The list of columns of `grouping_id` should match grouping columns (in `cube` or `rollup`) exactly, or empty which means all the grouping columns (which is exactly what the function expects).

Note	<code>grouping_id</code> can only be used with <code>cube</code> , <code>rollup</code> or <code>GROUPING SETS</code> multi-dimensional aggregate operators (and is verified when Analyzer does check analysis).
------	---

Note	Spark SQL's <code>grouping_id</code> function is known as <code>grouping_id</code> in Hive.
------	---

From [Hive's documentation about Grouping_ID function](#):

When aggregates are displayed for a column its value is `null`. This may conflict in case the column itself has some `null` values. There needs to be some way to identify `NULL` in column, which means aggregate and `NULL` in column, which means value. `GROUPING_ID` function is the solution to that.

Internally, `grouping_id()` creates a [Column](#) with `GroupingID` unevaluable expression.

Note	Unevaluable expressions are expressions replaced by some other expressions during analysis or optimization .
------	--

```
// workshops dataset was defined earlier
val q = workshops
  .cube("city", "year")
  .agg(grouping_id())

// grouping_id function is spark_grouping_id virtual column internally
// that is resolved during analysis - see Analyzed Logical Plan
scala> q.explain(true)
== Parsed Logical Plan ==
'Aggregate [cube(city#182, year#183)], [city#182, year#183, grouping_id() AS grouping_
```

```

id()#742]
+- Union
  :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
    :  +- LocalRelation [_1#178, _2#179, _3#180]
  +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
    +- LocalRelation [_1#192, _2#193, _3#194]

== Analyzed Logical Plan ==
city: string, year: int, grouping_id(): int
Aggregate [city#757, year#758, spark_grouping_id#754], [city#757, year#758, spark_grouping_id#754 AS grouping_id()#742]
+- Expand [List(city#182, year#183, count#184, city#755, year#756, 0), List(city#182, year#183, count#184, city#755, null, 1), List(city#182, year#183, count#184, null, year#756, 2), List(city#182, year#183, count#184, null, null, 3)], [city#182, year#183, count#184, city#757, year#758, spark_grouping_id#754]
  +- Project [city#182, year#183, count#184, city#182 AS city#755, year#183 AS year#756]
    +- Union
      :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
        :  +- LocalRelation [_1#178, _2#179, _3#180]
      +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
        +- LocalRelation [_1#192, _2#193, _3#194]

== Optimized Logical Plan ==
Aggregate [city#757, year#758, spark_grouping_id#754], [city#757, year#758, spark_grouping_id#754 AS grouping_id()#742]
+- Expand [List(city#755, year#756, 0), List(city#755, null, 1), List(null, year#756, 2), List(null, null, 3)], [city#757, year#758, spark_grouping_id#754]
  +- Union
    :- LocalRelation [city#755, year#756]
    +- LocalRelation [city#755, year#756]

== Physical Plan ==
*HashAggregate(keys=[city#757, year#758, spark_grouping_id#754], functions=[], output=[city#757, year#758, grouping_id()#742])
+- Exchange hashpartitioning(city#757, year#758, spark_grouping_id#754, 200)
  +- *HashAggregate(keys=[city#757, year#758, spark_grouping_id#754], functions=[], output=[city#757, year#758, spark_grouping_id#754])
    +- *Expand [List(city#755, year#756, 0), List(city#755, null, 1), List(null, year#756, 2), List(null, null, 3)], [city#757, year#758, spark_grouping_id#754]
      +- Union
        :- LocalTableScan [city#755, year#756]
        +- LocalTableScan [city#755, year#756]

```

Note

`grouping_id` was added to Spark SQL in [\[SPARK-12706\]](#) support `grouping/grouping_id` function together group set.

Standard Functions for Collections (Collection Functions)

Table 1. (Subset of) Standard Functions for Handling Collections

Name	Description
array_contains	array_contains(column: Column, value: Any): Column
explode	explode(e: Column): Column
explode_outer	explode_outer(e: Column): Column Creates a new row for each element in the given array or map column. If the array/map is <code>null</code> or empty then <code>null</code> is produced.
from_json	from_json(e: Column, schema: DataType): Column from_json(e: Column, schema: DataType, options: Map[String, String]): Column from_json(e: Column, schema: String, options: Map[String, String]): Column from_json(e: Column, schema: StructType): Column from_json(e: Column, schema: StructType, options: Map[String, String]): Column
map_keys	map_keys(e: Column): Column
map_values	map_values(e: Column): Column
posexplode	posexplode(e: Column): Column
posexplode_outer	posexplode_outer(e: Column): Column

	reverse(e: Column): Column		
reverse	Returns a reversed string or an array with reverse order of elements		
	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px; width: 15%;">Note</td> <td style="padding: 2px 10px;">Support for reversing arrays is new in 2.4.0.</td> </tr> </table>	Note	Support for reversing arrays is new in 2.4.0 .
Note	Support for reversing arrays is new in 2.4.0 .		

	size(e: Column): Column
size	Returns the size of the given array or map. Returns -1 if <code>null</code> .

reverse Collection Function

```
reverse(e: Column): Column
```

reverse ...FIXME

size Collection Function

```
size(e: Column): Column
```

`size` returns the size of the given array or map. Returns -1 if `null`.

Internally, `size` creates a `column` with `Size` unary expression.

```
import org.apache.spark.sql.functions.size
val c = size('id)
scala> println(c.expr.asCode)
Size(UnresolvedAttribute(ArrayBuffer(id)))
```

posexplode Collection Function

```
posexplode(e: Column): Column
```

posexplode ...FIXME

posexplode_outer Collection Function

```
posexplode_outer(e: Column): Column
```

```
poseplode_outer ...FIXME
```

explode Collection Function

Caution	FIXME
---------	-------

```
scala> Seq(Array(0,1,2)).toDF("array").withColumn("num", explode('array)).show
+-----+---+
|    array|num|
+-----+---+
|[0, 1, 2]|  0|
|[0, 1, 2]|  1|
|[0, 1, 2]|  2|
+-----+---+
```

Note	explode function is an equivalent of <code>flatMap</code> operator for <code>Dataset</code> .
------	---

explode_outer Collection Function

```
explode_outer(e: Column): Column
```

`explode_outer` generates a new row for each element in `e` array or map column.

Note	Unlike <code>explode</code> , <code>explode_outer</code> generates <code>null</code> when the array or map is <code>null</code> or empty.
------	---

```
val arrays = Seq((1,Seq.empty[String])).toDF("id", "array")
scala> arrays.printSchema
root
|-- id: integer (nullable = false)
|-- array: array (nullable = true)
|   |-- element: string (containsNull = true)
scala> arrays.select(explode_outer($"array")).show
+---+
| col|
+---+
| null|
+---+
```

Internally, `explode_outer` creates a `Column` with `GeneratorOuter` and `Explode Catalyst` expressions.

```

val explodeOuter = explode_outer($"array").expr
scala> println(explodeOuter.numberedTreeString)
00 generatorouter(explode('array))
01 +- explode('array)
02   +- 'array

```

Extracting Data from Arbitrary JSON-Encoded Values — `from_json` Collection Function

```

from_json(e: Column, schema: StructType, options: Map[String, String]): Column (1)
from_json(e: Column, schema: DataType, options: Map[String, String]): Column (2)
from_json(e: Column, schema: StructType): Column (3)
from_json(e: Column, schema: DataType): Column (4)
from_json(e: Column, schema: String, options: Map[String, String]): Column (5)

```

1. Calls <2> with `StructType` converted to `DataType`
2. (fixme)
3. Calls <1> with empty `options`
4. Relays to the other `from_json` with empty `options`
5. Uses schema as `DataType` in the JSON format or falls back to `StructType` in the DDL format

`from_json` parses a column with a JSON-encoded value into a [StructType](#) or [ArrayType](#) of `StructType` elements with the specified schema.

```

val jsons = Seq("""{ "id": 0 }""").toDF("json")

import org.apache.spark.sql.types._
val schema = new StructType()
  .add($"id".int.copy(nullable = false))

import org.apache.spark.sql.functions.from_json
scala> jsons.select(from_json($"json", schema) as "ids").show
+---+
|ids|
+---+
|[0]|
+---+

```

Note

- A schema can be one of the following:
1. [DataType](#) as a Scala object or in the JSON format
 2. [StructType](#) in the DDL format

```
// Define the schema for JSON-encoded messages
// Note that the schema is nested (on the addresses field)
import org.apache.spark.sql.types._

val addressesSchema = new StructType()
  .add($"city".string)
  .add($"state".string)
  .add($"zip".string)

val schema = new StructType()
  .add($"firstName".string)
  .add($"lastName".string)
  .add($"email".string)
  .add($"addresses".array(addressesSchema))

scala> schema.printTreeString
root
|-- firstName: string (nullable = true)
|-- lastName: string (nullable = true)
|-- email: string (nullable = true)
|-- addresses: array (nullable = true)
|   |-- element: struct (containsNull = true)
|   |   |-- city: string (nullable = true)
|   |   |-- state: string (nullable = true)
|   |   |-- zip: string (nullable = true)

// Generate the JSON-encoded schema
// That's the variant of the schema that from_json accepts
val schemaAsJson = schema.json

// Use prettyJson to print out the JSON-encoded schema
// Only for demo purposes
scala> println(schema.prettyJson)
{
  "type" : "struct",
  "fields" : [ {
    "name" : "firstName",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "lastName",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "email",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  } ]
}
```

```

    "nullable" : true,
    "metadata" : { }
}, {
  "name" : "addresses",
  "type" : {
    "type" : "array",
    "elementType" : {
      "type" : "struct",
      "fields" : [ {
        "name" : "city",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "state",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "zip",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      } ]
    },
    "containsNull" : true
  },
  "nullable" : true,
  "metadata" : { }
} ]
}

// Let's "validate" the JSON-encoded schema
import org.apache.spark.sql.types.DataType
val dt = DataType.fromJson(schemaAsJson)
scala> println(dt.sql)
STRUCT<'firstName`: STRING, `lastName`: STRING, `email`: STRING, `addresses`: ARRAY<STRUCT<'city`: STRING, `state`: STRING, `zip`: STRING>>>

// No exception means that the JSON-encoded schema should be fine
// Use it with from_json
val rawJsons = Seq("""
{
  "firstName" : "Jacek",
  "lastName" : "Laskowski",
  "email" : "jacek@japila.pl",
  "addresses" : [
    {
      "city" : "Warsaw",
      "state" : "N/A",
      "zip" : "02-791"
    }
  ]
}

```

```

    }
""").toDF("rawjson")
val people = rawJsons
  .select(from_json($"rawjson", schemaAsJson, Map.empty[String, String]) as "json")
  .select("json.*") // <-- flatten the struct field
  .withColumn("address", explode($"addresses")) // <-- explode the array field
  .drop("addresses") // <-- no longer needed
  .select("firstName", "lastName", "email", "address.*") // <-- flatten the struct file
ld
scala> people.show
+-----+-----+-----+-----+
|firstName| lastName|      email| city|state|   zip|
+-----+-----+-----+-----+
|     Jacek|Laskowski|jacek@japila.pl|Warsaw| N/A|02-791|
+-----+-----+-----+-----+

```

Note `options` controls how a JSON is parsed and contains the same options as the [json](#) format.

Internally, `from_json` creates a [Column](#) with [JsonToStructs](#) unary expression.

Note `from_json` (creates a [JsonToStructs](#) that) uses a JSON parser in [FAILFAST](#) parsing mode that simply fails early when a corrupted/malformed record is found (and hence does not support `columnNameOfCorruptRecord` JSON option).

```

val jsons = Seq("""{ id: 0 }""").toDF("json")

import org.apache.spark.sql.types._
val schema = new StructType()
  .add($"id".int.copy(nullable = false))
  .add($"corrupted_records".string)
val opts = Map("columnNameOfCorruptRecord" -> "corrupted_records")
scala> jsons.select(from_json($"json", schema, opts) as "ids").show
+---+
| ids|
+---+
| null|
+---+

```

Note `from_json` corresponds to SQL's `from_json`.

array_contains Collection Function

```
array_contains(column: Column, value: Any): Column
```

`array_contains` creates a `Column` for a `Column` argument as an `array` and the `value` of same type as the type of the elements of the array.

Internally, `array_contains` creates a `Column` with a `ArrayContains` expression.

```
// Arguments must be an array followed by a value of same type as the array elements
import org.apache.spark.sql.functions.array_contains
val c = array_contains(column = $"ids", value = 1)

val ids = Seq(Seq(1,2,3), Seq(1), Seq(2,3)).toDF("ids")
val q = ids.filter(c)
scala> q.show
+-----+
|     ids|
+-----+
|[1, 2, 3]|
|      [1]|
+-----+
```

`array_contains` corresponds to SQL's `array_contains`.

```
import org.apache.spark.sql.functions.array_contains
val c = array_contains(column = $"ids", value = Array(1, 2))
val e = c.expr
scala> println(e.sql)
array_contains(`ids`, [1,2])
```

Tip

Use SQL's `array_contains` to use values from columns for the `column` and `value` arguments.

```

val codes = Seq(
  (Seq(1, 2, 3), 2),
  (Seq(1), 1),
  (Seq.empty[Int], 1),
  (Seq(2, 4, 6), 0)).toDF("codes", "cd")
scala> codes.show
+-----+---+
|    codes| cd|
+-----+---+
|[1, 2, 3]|  2|
|[1]|  1|
|[]|  1|
|[2, 4, 6]|  0|
+-----+---+

val q = codes.where("array_contains(codes, cd)")
scala> q.show
+-----+---+
|    codes| cd|
+-----+---+
|[1, 2, 3]|  2|
|[1]|  1|
+-----+---+

// array_contains standard function with Columns does NOT work. Why?!
// Asked this question on StackOverflow --> https://stackoverflow.com/q/50412939/1305344
val q = codes.where(array_contains($"codes", $"cd"))
scala> q.show
java.lang.RuntimeException: Unsupported literal type class org.apache.spark.sql.Column
Name cd
  at org.apache.spark.sql.catalyst.expressions.Literal$.apply(literals.scala:77)
  at org.apache.spark.sql.functions.array_contains(functions.scala:3046)
  ... 50 elided

// Thanks Russel for this excellent "workaround"
// https://stackoverflow.com/a/50413766/1305344
import org.apache.spark.sql.Column
import org.apache.spark.sql.catalyst.expressions.ArrayContains
val q = codes.where(new Column(ArrayContains($"codes".expr, $"cd".expr)))
scala> q.show
+-----+---+
|    codes| cd|
+-----+---+
|[1, 2, 3]|  2|
|[1]|  1|
+-----+---+

```

map_keys Collection Function

```
map_keys(e: Column): Column
```

```
map_keys ...FIXME
```

map_values Collection Function

```
map_values(e: Column): Column
```

```
map_values ...FIXME
```

Date and Time Functions

Table 1. (Subset of) Standard Functions for Date and Time

Name	Description
<code>current_date</code>	Gives current date as a date column
<code>current_timestamp</code>	
<code>date_format</code>	
<code>to_date</code>	Converts column to date type (with an optional date format)
<code>to_timestamp</code>	Converts column to timestamp type (with an optional timestamp format)
<code>unix_timestamp</code>	Converts current or specified time to Unix timestamp (in seconds)
<code>window</code>	Generates time windows (i.e. tumbling, sliding and delayed windows)

Current Date As Date Column — `current_date` Function

```
current_date(): Column
```

`current_date` function gives the current date as a `date` column.

```
val df = spark.range(1).select(current_date)
scala> df.show
+-----+
|current_date()|
+-----+
|  2017-09-16|
+-----+

scala> df.printSchema
root
 |-- current_date(): date (nullable = false)
```

Internally, `current_date` creates a `Column` with `CurrentDate` Catalyst leaf expression.

```

val c = current_date()
import org.apache.spark.sql.catalyst.expressions.CurrentDate
val cd = c.expr.asInstanceOf[CurrentDate]
scala> println(cd.prettyName)
current_date

scala> println(cd.numberedTreeString)
00 current_date(None)

```

date_format Function

```
date_format(dateExpr: Column, format: String): Column
```

Internally, `date_format` creates a `Column` with `DateFormatClass` binary expression.

`DateFormatClass` takes the expression from `dateExpr` column and `format`.

```

val c = date_format($"date", "dd/MM/yyyy")

import org.apache.spark.sql.catalyst.expressions.DateFormatClass
val dfc = c.expr.asInstanceOf[DateFormatClass]
scala> println(dfc.prettyName)
date_format

scala> println(dfc.numberedTreeString)
00 date_format('date, dd/MM/yyyy, None)
01 :- 'date
02 +- dd/MM/yyyy

```

current_timestamp Function

```
current_timestamp(): Column
```

Caution	FIXME
Note	<code>current_timestamp</code> is also <code>now</code> function in SQL.

Converting Current or Specified Time to Unix Timestamp — unix_timestamp Function

```
unix_timestamp(): Column (1)
unix_timestamp(time: Column): Column (2)
unix_timestamp(time: Column, format: String): Column
```

1. Gives current timestamp (in seconds)
2. Converts `time` string in format `yyyy-MM-dd HH:mm:ss` to Unix timestamp (in seconds)

`unix_timestamp` converts the current or specified `time` in the specified `format` to a Unix timestamp (in seconds).

`unix_timestamp` supports a column of type `Date`, `Timestamp` or `String`.

```
// no time and format => current time
scala> spark.range(1).select(unix_timestamp as "current_timestamp").show
+-----+
|current_timestamp|
+-----+
|      1493362850|
+-----+

// no format so yyyy-MM-dd HH:mm:ss assumed
scala> Seq("2017-01-01 00:00:00").toDF("time").withColumn("unix_timestamp", unix_timestamp($"time")).show
+-----+-----+
|       time|unix_timestamp|
+-----+-----+
|2017-01-01 00:00:00|     1483225200|
+-----+-----+

scala> Seq("2017/01/01 00:00:00").toDF("time").withColumn("unix_timestamp", unix_timestamp($"time", "yyyy/MM/dd")).show
+-----+-----+
|       time|unix_timestamp|
+-----+-----+
|2017/01/01 00:00:00|     1483225200|
+-----+-----+
```

`unix_timestamp` returns `null` if conversion fails.

```
// note slashes as date separators
scala> Seq("2017/01/01 00:00:00").toDF("time").withColumn("unix_timestamp", unix_timestamp($"time")).show
+-----+-----+
|       time|unix_timestamp|
+-----+-----+
|2017/01/01 00:00:00|         null|
+-----+-----+
```

Note

`unix_timestamp` is also supported in [SQL mode](#).

```
scala> spark.sql("SELECT unix_timestamp() as unix_timestamp").show
+-----+
| unix_timestamp |
+-----+
| 1493369225 |
+-----+
```

Internally, `unix_timestamp` creates a [Column](#) with [UnixTimestamp](#) binary expression (possibly with `currentTimestamp`).

Generating Time Windows — `window` Function

```
window(
  timeColumn: Column,
  windowDuration: String): Column (1)
window(
  timeColumn: Column,
  windowDuration: String,
  slideDuration: String): Column (2)
window(
  timeColumn: Column,
  windowDuration: String,
  slideDuration: String,
  startTime: String): Column (3)
```

1. Creates a tumbling time window with `slideDuration` as `windowDuration` and `0` second for `startTime`
2. Creates a sliding time window with `0` second for `startTime`
3. Creates a delayed time window

`window` generates **tumbling**, **sliding** or **delayed** time windows of `windowDuration` duration given a `timeColumn` timestamp specifying column.

Note

From [Tumbling Window \(Azure Stream Analytics\)](#):

Tumbling windows are a series of fixed-sized, non-overlapping and contiguous time intervals.

Note

From [Introducing Stream Windows in Apache Flink](#):

Tumbling windows group elements of a stream into finite sets where each set corresponds to an interval.

Tumbling windows discretize a stream into non-overlapping windows.

```
scala> val timeColumn = window('time, "5 seconds")
timeColumn: org.apache.spark.sql.Column = timewindow(time, 5000000, 5000000, 0) AS `window`
```

`timeColumn` should be of [TimestampType](#), i.e. with `java.sql.Timestamp` values.

Tip

Use `java.sql.Timestamp.from` or `java.sql.Timestamp.valueOf` factory methods to create `Timestamp` instances.

```
// https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html
import java.time.LocalDateTime
// https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html
import java.sql.Timestamp
val levels = Seq(
    // (year, month, dayOfMonth, hour, minute, second)
    ((2012, 12, 12, 12, 12, 12), 5),
    ((2012, 12, 12, 12, 12, 14), 9),
    ((2012, 12, 12, 13, 13, 14), 4),
    ((2016, 8, 13, 0, 0, 0), 10),
    ((2017, 5, 27, 0, 0, 0), 15)).
    map { case ((yy, mm, dd, h, m, s), a) => (LocalDateTime.of(yy, mm, dd, h, m, s), a) }.
).map { case (ts, a) => (Timestamp.valueOf(ts), a) }.
toDF("time", "level")
scala> levels.show
+-----+-----+
|          time|level|
+-----+-----+
|2012-12-12 12:12:12|     5|
|2012-12-12 12:12:14|     9|
|2012-12-12 13:13:14|     4|
|2016-08-13 00:00:00|    10|
|2017-05-27 00:00:00|    15|
+-----+-----+

val q = levels.select(window($"time", "5 seconds"), $"level")
scala> q.show(truncate = false)
+-----+-----+
|window                               |level|
+-----+-----+
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|5    |
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|9    |
|[2012-12-12 13:13:10.0,2012-12-12 13:13:15.0]|4    |
```

```

|[2016-08-13 00:00:00.0,2016-08-13 00:00:05.0]|10    |
|[2017-05-27 00:00:00.0,2017-05-27 00:00:05.0]|15    |
+-----+-----+-----+-----+-----+-----+-----+-----+
scala> q.printSchema
root
|-- window: struct (nullable = true)
|   |-- start: timestamp (nullable = true)
|   |-- end: timestamp (nullable = true)
|-- level: integer (nullable = false)

// calculating the sum of levels every 5 seconds
val sums = levels.
  groupBy(window($"time", "5 seconds")).
  agg(sum("level") as "level_sum").
  select("window.start", "window.end", "level_sum")
scala> sums.show
+-----+-----+-----+-----+
|       start|           end|level_sum|
+-----+-----+-----+-----+
|2012-12-12 13:13:10|2012-12-12 13:13:15|      4|
|2012-12-12 12:12:10|2012-12-12 12:12:15|     14|
|2016-08-13 00:00:00|2016-08-13 00:00:05|     10|
|2017-05-27 00:00:00|2017-05-27 00:00:05|     15|
+-----+-----+-----+-----+

```

`windowDuration` and `slideDuration` are strings specifying the width of the window for duration and sliding identifiers, respectively.

Tip	Use <code>CalendarInterval</code> for valid window identifiers.
-----	---

Note	<code>window</code> is available as of Spark 2.0.0 .
------	---

Internally, `window` creates a `Column` (with `TimeWindow` expression) available as `window` alias.

```
// q is the query defined earlier
scala> q.show(truncate = false)
+-----+-----+
|window |level|
+-----+-----+
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|5   |
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|9   |
|[2012-12-12 13:13:10.0,2012-12-12 13:13:15.0]|4   |
|[2016-08-13 00:00:00.0,2016-08-13 00:00:05.0]|10  |
|[2017-05-27 00:00:00.0,2017-05-27 00:00:05.0]|15  |
+-----+-----+

scala> println(timeColumn.expr.numberedTreeString)
00 timewindow('time, 5000000, 5000000, 0) AS window#22
01 +- timewindow('time, 5000000, 5000000, 0)
02   +- 'time
```

Example — Traffic Sensor

Note	The example is borrowed from Introducing Stream Windows in Apache Flink .
------	---

The example shows how to use `window` function to model a traffic sensor that counts every 15 seconds the number of vehicles passing a certain location.

Converting Column To DateType — `to_date` Function

```
to_date(e: Column): Column
to_date(e: Column, fmt: String): Column
```

`to_date` converts the column into `DateType` (by casting to `DateType`).

Note	<code>fmt</code> follows the formatting styles .
------	--

Internally, `to_date` creates a `Column` with `ParseToDate` expression (and `Literal` expression for `fmt`).

Tip	Use <code>ParseToDate</code> expression to use a column for the values of <code>fmt</code> .
-----	--

Converting Column To TimestampType — `to_timestamp` Function

```
to_timestamp(s: Column): Column
to_timestamp(s: Column, fmt: String): Column
```

`to_timestamp` converts the column into [TimestampType](#) (by casting to `TimestampType`).

Note

`fmt` follows [the formatting styles](#).

Internally, `to_timestamp` creates a [Column](#) with [ParseToTimestamp](#) expression (and [Literal](#) expression for `fmt`).

Tip

Use [ParseToTimestamp](#) expression to use a column for the values of `fmt` .

Regular Functions (Non-Aggregate Functions)

Table 1. (Subset of) Regular Functions

Name	Description
array	
broadcast	
coalesce	Gives the first non- <code>null</code> value among the given columns or <code>null</code> .
col and column	Creating Columns
expr	
lit	
map	
monotonically_increasing_id	
struct	
typedLit	
when	

broadcast Function

```
broadcast[T](df: Dataset[T]): Dataset[T]
```

`broadcast` function marks the input [Dataset](#) as small enough to be used in broadcast join.

Tip	Read up on Broadcast Joins (aka Map-Side Joins) .
-----	---

```

val left = Seq((0, "aa"), (0, "bb")).toDF("id", "token").as[(Int, String)]
val right = Seq(("aa", 0.99), ("bb", 0.57)).toDF("token", "prob").as[(String, Double)]

scala> left.join(broadcast(right), "token").explain(extended = true)
== Parsed Logical Plan ==
'Join UsingJoin(Inner,List(token))
:- Project [_1#123 AS id#126, _2#124 AS token#127]
: +- LocalRelation [_1#123, _2#124]
+- BroadcastHint
  +- Project [_1#136 AS token#139, _2#137 AS prob#140]
    +- LocalRelation [_1#136, _2#137]

== Analyzed Logical Plan ==
token: string, id: int, prob: double
Project [token#127, id#126, prob#140]
+- Join Inner, (token#127 = token#139)
  :- Project [_1#123 AS id#126, _2#124 AS token#127]
  : +- LocalRelation [_1#123, _2#124]
  +- BroadcastHint
    +- Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- LocalRelation [_1#136, _2#137]

== Optimized Logical Plan ==
Project [token#127, id#126, prob#140]
+- Join Inner, (token#127 = token#139)
  :- Project [_1#123 AS id#126, _2#124 AS token#127]
  : +- Filter isnotnull(_2#124)
  :   +- LocalRelation [_1#123, _2#124]
  +- BroadcastHint
    +- Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- Filter isnotnull(_1#136)
        +- LocalRelation [_1#136, _2#137]

== Physical Plan ==
*Project [token#127, id#126, prob#140]
+- *BroadcastHashJoin [token#127], [token#139], Inner, BuildRight
  :- *Project [_1#123 AS id#126, _2#124 AS token#127]
  : +- *Filter isnotnull(_2#124)
  :   +- LocalTableScan [_1#123, _2#124]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
    +- *Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- *Filter isnotnull(_1#136)
        +- LocalTableScan [_1#136, _2#137]

```

Note

`broadcast` standard function is a special case of `Dataset.hint` operator that allows for attaching any hint to a logical plan.

coalesce Function

```
coalesce(e: Column*): Column
```

`coalesce` gives the first non-`null` value among the given columns or `null`.

`coalesce` requires at least one column and all columns have to be of the same or compatible types.

Internally, `coalesce` creates a `Column` with a `Coalesce` expression (with the children being the `expressions` of the input `Column`).

Example: `coalesce` Function

```
val q = spark.range(2)
    .select(
      coalesce(
        lit(null),
        lit(null),
        lit(2) + 2,
        $"id" as "first non-null value")
    )
scala> q.show
+-----+
|first non-null value|
+-----+
|          4|
|          4|
+-----+
```

Creating Columns—`col` and `column` Functions

```
col(colName: String): Column
column(colName: String): Column
```

`col` and `column` methods create a `Column` that you can later use to reference a column in a dataset.

```
import org.apache.spark.sql.functions._

scala> val nameCol = col("name")
nameCol: org.apache.spark.sql.Column = name

scala> val cityCol = column("city")
cityCol: org.apache.spark.sql.Column = city
```

expr Function

```
expr(expr: String): Column
```

`expr` function parses the input `expr` SQL statement to a `Column` it represents.

```
val ds = Seq((0, "hello"), (1, "world"))
  .toDF("id", "token")
  .as[(Long, String)]  
  

scala> ds.show
+---+-----+
| id|token|
+---+-----+
|  0|hello|
|  1|world|
+---+-----+  
  

val filterExpr = expr("token = 'hello'")  
  

scala> ds.filter(filterExpr).show
+---+-----+
| id|token|
+---+-----+
|  0|hello|
+---+-----+
```

Internally, `expr` uses the active session's `sqlParser` or creates a new `SparkSqlParser` to call `parseExpression` method.

lit Function

```
lit(literal: Any): Column
```

`lit` function...FIXME

struct Functions

```
struct(cols: Column*): Column
struct(colName: String, colNames: String*): Column
```

`struct` family of functions allows you to create a new struct column based on a collection of `Column` or their names.

Note

The difference between `struct` and another similar `array` function is that the types of the columns can be different (in `struct`).

```
scala> df.withColumn("struct", struct($"name", $"val")).show
+---+---+-----+
| id|val| name|  struct|
+---+---+-----+
|  0|  1|hello|[hello,1]|
|  2|  3|world|[world,3]|
|  2|  4| ala| [ala,4]|
+---+---+-----+
```

typedLit Function

```
typedLit[T : TypeTag](literal: T): Column
```

`typedLit` ...FIXME

array Function

```
array(cols: Column*): Column
array(colName: String, colNames: String*): Column
```

`array` ...FIXME

map Function

```
map(cols: Column*): Column
```

`map` ...FIXME

when Function

```
when(condition: Column, value: Any): Column
```

`when` ...FIXME

monotonically_increasing_id Function

```
monotonically_increasing_id(): Column
```

`monotonically_increasing_id` returns monotonically increasing 64-bit integers. The generated IDs are guaranteed to be monotonically increasing and unique, but not consecutive (unless all rows are in the same single partition which you rarely want due to the amount of the data).

```
val q = spark.range(1).select(monotonically_increasing_id)
scala> q.show
+-----+
|monotonically_increasing_id()|
+-----+
|          60129542144|
+-----+
```

The [current implementation](#) uses the partition ID in the upper 31 bits, and the lower 33 bits represent the record number within each partition. That assumes that the data set has less than 1 billion partitions, and each partition has less than 8 billion records.

```
// Demo to show the internals of monotonically_increasing_id function
// i.e. how MonotonicallyIncreasingID expression works

// Create a dataset with the same number of rows per partition
val q = spark.range(start = 0, end = 8, step = 1, numPartitions = 4)

// Make sure that every partition has the same number of rows
q.mapPartitions(rows => Iterator(rows.size)).foreachPartition(rows => assert(rows.next
  == 2))
q.select(monotonically_increasing_id).show

// Assign consecutive IDs for rows per partition
import org.apache.spark.sql.expressions.Window
// count is the name of the internal registry of MonotonicallyIncreasingID to count ro
ws
// Could also be "id" since it is unique and consecutive in a partition
import org.apache.spark.sql.functions.{row_number, shiftLeft, spark_partition_id}
val rowNumber = row_number over Window.partitionBy(spark_partition_id).orderBy("id")
// row_number is a sequential number starting at 1 within a window partition
val count = rowNumber - 1 as "count"
val partitionMask = shiftLeft(spark_partition_id cast "long", 33) as "partitionMask"
// FIXME Why does the following sum give "weird" results?!
val sum = (partitionMask + count) as "partitionMask + count"
val demo = q.select(
  $"id",
  partitionMask,
  count,
  // FIXME sum,
  monotonically_increasing_id)
scala> demo.orderBy("id").show
+---+-----+-----+-----+
| id|partitionMask|count|monotonically_increasing_id()|
+---+-----+-----+-----+
|  0|          0|    0|                  0|
|  1|          0|    1|                  1|
|  2| 8589934592|    0| 8589934592|
|  3| 8589934592|    1| 8589934593|
|  4| 17179869184|    0| 17179869184|
|  5| 17179869184|    1| 17179869185|
|  6| 25769803776|    0| 25769803776|
|  7| 25769803776|    1| 25769803777|
+---+-----+-----+-----+
```

Internally, `monotonically_increasing_id` creates a [Column](#) with a [MonotonicallyIncreasingID](#) non-deterministic leaf expression.

Standard Functions for Window Aggregation (Window Functions)

Window aggregate functions (aka **window functions** or **windowed aggregates**) are functions that perform a calculation over a group of records called **window** that are in *some* relation to the current record (i.e. can be in the same partition or frame as the current row).

In other words, when executed, a window function computes a value for each and every row in a window (per [window specification](#)).

Note	Window functions are also called over functions due to how they are applied using over operator.
------	---

Spark SQL supports three kinds of window functions:

- **ranking** functions
- **analytic** functions
- **aggregate** functions

Table 1. Window Aggregate Functions in Spark SQL

	Function	Purpose
Ranking functions	rank	
	dense_rank	
	percent_rank	
	ntile	
	row_number	
Analytic functions	cume_dist	
	lag	
	lead	

For aggregate functions, you can use the existing [aggregate functions](#) as window functions, e.g. `sum` , `avg` , `min` , `max` and `count` .

```
// Borrowed from 3.5. Window Functions in PostgreSQL documentation
// Example of window functions using Scala API
//

case class Salary(depName: String, empNo: Long, salary: Long)
val empsalary = Seq(
  Salary("sales", 1, 5000),
  Salary("personnel", 2, 3900),
  Salary("sales", 3, 4800),
  Salary("sales", 4, 4800),
  Salary("personnel", 5, 3500),
  Salary("develop", 7, 4200),
  Salary("develop", 8, 6000),
  Salary("develop", 9, 4500),
  Salary("develop", 10, 5200),
  Salary("develop", 11, 5200)).toDS

import org.apache.spark.sql.expressions.Window
// Windows are partitions of depName
scala> val byDepName = Window.partitionBy('depName)
byDepName: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@1a711314

scala> empsalary.withColumn("avg", avg('salary) over byDepName).show
+-----+-----+-----+
| depName|empNo|salary|           avg|
+-----+-----+-----+
| develop|    7|  4200|      5020.0|
| develop|    8|  6000|      5020.0|
| develop|    9|  4500|      5020.0|
| develop|   10|  5200|      5020.0|
| develop|   11|  5200|      5020.0|
| sales|    1|  5000| 4866.666666666667|
| sales|    3|  4800| 4866.666666666667|
| sales|    4|  4800| 4866.666666666667|
| personnel|  2|  3900|      3700.0|
| personnel|  5|  3500|      3700.0|
+-----+-----+-----+
```

You describe a window using the convenient factory methods in [Window object](#) that create a [window specification](#) that you can further refine with **partitioning**, **ordering**, and **frame boundaries**.

After you describe a window you can apply [window aggregate functions](#) like **ranking** functions (e.g. `RANK`), **analytic** functions (e.g. `LAG`), and the regular [aggregate functions](#), e.g. `sum`, `avg`, `max`.

Note

Window functions are supported in structured queries using [SQL](#) and [Column-based expressions](#).

Although similar to [aggregate functions](#), a window function does not group rows into a single output row and retains their separate identities. A window function can access rows that are linked to the current row.

Note

The main difference between window aggregate functions and [aggregate functions](#) with [grouping operators](#) is that the former calculate values for every row in a window while the latter gives you at most the number of input rows, one value per group.

Tip

See [Examples](#) section in this document.

You can mark a function `window` by `OVER` clause after a function in SQL, e.g. `avg(revenue) OVER (...)` or [over method](#) on a function in the Dataset API, e.g. `rank().over(...)`.

Note

Window functions belong to [Window functions group](#) in Spark's Scala API.

Note

Window-based framework is available as an experimental feature since Spark **1.4.0**.

Window object

`Window` object provides functions to define windows (as [WindowSpec instances](#)).

`Window` object lives in `org.apache.spark.sql.expressions` package. Import it to use `Window` functions.

```
import org.apache.spark.sql.expressions.Window
```

There are two families of the functions available in `Window` object that create [WindowSpec](#) instance for one or many [Column](#) instances:

- [partitionBy](#)
- [orderBy](#)

Partitioning Records — `partitionBy` Methods

```
partitionBy(colName: String, colNames: String*): WindowSpec
partitionBy(cols: Column*): WindowSpec
```

`partitionBy` creates an instance of `WindowSpec` with partition expression(s) defined for one or more columns.

```
// partition records into two groups
// * tokens starting with "h"
// * others
val byHTokens = Window.partitionBy('token startsWith "h")  
  

// count the sum of ids in each group
val result = tokens.select('*', sum('id) over byHTokens as "sum over h tokens").orderBy(
'id)  
  

scala> .show
+---+-----+-----+
| id|token|sum over h tokens|
+---+-----+-----+
| 0|hello|      4|
| 1|henry|      4|
| 2| and|      2|
| 3|harry|      4|
+---+-----+-----+
```

Ordering in Windows — `orderBy` Methods

```
orderBy(colName: String, colNames: String*): WindowSpec
orderBy(cols: Column*): WindowSpec
```

`orderBy` allows you to control the order of records in a window.

```

import org.apache.spark.sql.expressions.Window
val byDepnameSalaryDesc = Window.partitionBy('depname).orderBy('salary desc)

// a numerical rank within the current row's partition for each distinct ORDER BY value

scala> val rankByDepname = rank().over(byDepnameSalaryDesc)
rankByDepname: org.apache.spark.sql.Column = RANK() OVER (PARTITION BY depname ORDER BY salary DESC UnspecifiedFrame)

scala> empsalary.select('*', rankByDepname as 'rank).show
+-----+-----+-----+
| depName|empNo|salary|rank|
+-----+-----+-----+
| develop|    8|  6000|    1|
| develop|   10|  5200|    2|
| develop|   11|  5200|    2|
| develop|    9|  4500|    4|
| develop|    7|  4200|    5|
| sales|    1|  5000|    1|
| sales|    3|  4800|    2|
| sales|    4|  4800|    2|
| personnel|   2|  3900|    1|
| personnel|   5|  3500|    2|
+-----+-----+-----+

```

rangeBetween Method

```
rangeBetween(start: Long, end: Long): WindowSpec
```

`rangeBetween` creates a `WindowSpec` with the frame boundaries from `start` (inclusive) to `end` (inclusive).

Note

It is recommended to use `window.unboundedPreceding`, `window.unboundedFollowing` and `window.currentRow` to describe the frame boundaries when a frame is unbounded preceding, unbounded following and at current row, respectively.

```

import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.WindowSpec
val spec: WindowSpec = Window.rangeBetween(Window.unboundedPreceding, Window.currentRow)

```

Internally, `rangeBetween` creates a `windowSpec` with `SpecifiedWindowFrame` and `RangeFrame` type.

Frame

At its core, a window function calculates a return value for every input row of a table based on a group of rows, called the **frame**. Every input row can have a unique frame associated with it.

When you define a frame you have to specify three components of a frame specification - the **start and end boundaries**, and the **type**.

Types of boundaries (two positions and three offsets):

- `UNBOUNDED PRECEDING` - the first row of the partition
- `UNBOUNDED FOLLOWING` - the last row of the partition
- `CURRENT ROW`
- `<value> PRECEDING`
- `<value> FOLLOWING`

Offsets specify the offset from the current input row.

Types of frames:

- `ROW` - based on *physical offsets* from the position of the current input row
- `RANGE` - based on *logical offsets* from the position of the current input row

In the current implementation of [WindowSpec](#) you can use two methods to define a frame:

- `rowsBetween`
- `rangeBetween`

See [WindowSpec](#) for their coverage.

Window Operators in SQL Queries

The grammar of windows operators in SQL accepts the following:

1. `CLUSTER BY` or `PARTITION BY` or `DISTRIBUTE BY` for partitions,
2. `ORDER BY` or `SORT BY` for sorting order,
3. `RANGE` , `ROWS` , `RANGE BETWEEN` , and `ROWS BETWEEN` for window frame types,
4. `UNBOUNDED PRECEDING` , `UNBOUNDED FOLLOWING` , `CURRENT ROW` for frame bounds.

Tip	Consult withWindows helper in <code>AstBuilder</code> .
-----	---

Examples

Top N per Group

Top N per Group is useful when you need to compute the first and second best-sellers in category.

Note

This example is borrowed from an [excellent article Introducing Window Functions in Spark SQL](#).

Table 2. Table PRODUCT_REVENUE

product	category	revenue
Thin	cell phone	6000
Normal	tablet	1500
Mini	tablet	5500
Ultra thin	cell phone	5000
Very thin	cell phone	6000
Big	tablet	2500
Bendable	cell phone	3000
Foldable	cell phone	3000
Pro	tablet	4500
Pro2	tablet	6500

Question: What are the best-selling and the second best-selling products in every category?

```

val dataset = Seq(
  ("Thin",      "cell phone", 6000),
  ("Normal",    "tablet",     1500),
  ("Mini",      "tablet",     5500),
  ("Ultra thin", "cell phone", 5000),
  ("Very thin",  "cell phone", 6000),
  ("Big",        "tablet",     2500),
  ("Bendable",   "cell phone", 3000),
  ("Foldable",   "cell phone", 3000),
  ("Pro",        "tablet",     4500),
  ("Pro2",       "tablet",     6500))
  .toDF("product", "category", "revenue")

scala> dataset.show
+-----+-----+-----+
| product| category|revenue|
+-----+-----+-----+
|     Thin|cell phone| 6000|
|   Normal|    tablet| 1500|
|     Mini|    tablet| 5500|
|Ultra thin|cell phone| 5000|
| Very thin|cell phone| 6000|
|      Big|    tablet| 2500|
| Bendable|cell phone| 3000|
| Foldable|cell phone| 3000|
|       Pro|    tablet| 4500|
|     Pro2|    tablet| 6500|
+-----+-----+-----+

scala> data.where('category === "tablet").show
+-----+-----+
|product|category|revenue|
+-----+-----+
| Normal|    tablet| 1500|
|   Mini|    tablet| 5500|
|     Big|    tablet| 2500|
|     Pro|    tablet| 4500|
|   Pro2|    tablet| 6500|
+-----+-----+

```

The question boils down to ranking products in a category based on their revenue, and to pick the best selling and the second best-selling products based the ranking.

```

import org.apache.spark.sql.expressions.Window
val overCategory = Window.partitionBy('category).orderBy('revenue.desc)

val ranked = data.withColumn("rank", dense_rank.over(overCategory))

scala> ranked.show
+-----+-----+-----+----+
| product| category|revenue|rank|
+-----+-----+-----+----+
|    Pro2|   tablet|  6500|  1|
|     Mini|   tablet|  5500|  2|
|      Pro|   tablet|  4500|  3|
|      Big|   tablet|  2500|  4|
|  Normal|   tablet|  1500|  5|
|     Thin|cell phone|  6000|  1|
| Very thin|cell phone|  6000|  1|
|Ultra thin|cell phone|  5000|  2|
| Bendable|cell phone|  3000|  3|
| Foldable|cell phone|  3000|  3|
+-----+-----+-----+----+

scala> ranked.where('rank <= 2).show
+-----+-----+-----+----+
| product| category|revenue|rank|
+-----+-----+-----+----+
|    Pro2|   tablet|  6500|  1|
|     Mini|   tablet|  5500|  2|
|     Thin|cell phone|  6000|  1|
| Very thin|cell phone|  6000|  1|
|Ultra thin|cell phone|  5000|  2|
+-----+-----+-----+----+

```

Revenue Difference per Category

Note

This example is the 2nd example from an *excellent* article [Introducing Window Functions in Spark SQL](#).

```
import org.apache.spark.sql.expressions.Window
val reveDesc = Window.partitionBy('category).orderBy('revenue.desc)
val reveDiff = max('revenue).over(reveDesc) - 'revenue

scala> data.select('*', reveDiff as 'revenue_diff).show
+-----+-----+-----+
| product| category|revenue|revenue_diff|
+-----+-----+-----+
|    Pro2|   tablet|  6500|        0|
|     Mini|   tablet|  5500|      1000|
|      Pro|   tablet|  4500|      2000|
|      Big|   tablet|  2500|      4000|
|  Normal|   tablet|  1500|      5000|
|     Thin|cell phone| 6000|        0|
| Very thin|cell phone| 6000|        0|
|Ultra thin|cell phone| 5000|      1000|
| Bendable|cell phone| 3000|      3000|
| Foldable|cell phone| 3000|      3000|
+-----+-----+-----+
```

Difference on Column

Compute a difference between values in rows in a column.

```

val pairs = for {
  x <- 1 to 5
  y <- 1 to 2
} yield (x, 10 * x * y)
val ds = pairs.toDF("ns", "tens")

scala> ds.show
+---+---+
| ns|tens|
+---+---+
| 1| 10|
| 1| 20|
| 2| 20|
| 2| 40|
| 3| 30|
| 3| 60|
| 4| 40|
| 4| 80|
| 5| 50|
| 5| 100|
+---+---+

import org.apache.spark.sql.expressions.Window
val overNs = Window.partitionBy('ns).orderBy('tens)
val diff = lead('tens, 1).over(overNs)

scala> ds.withColumn("diff", diff - 'tens).show
+---+---+---+
| ns|tens|diff|
+---+---+---+
| 1| 10| 10|
| 1| 20|null|
| 3| 30| 30|
| 3| 60|null|
| 5| 50| 50|
| 5| 100|null|
| 4| 40| 40|
| 4| 80|null|
| 2| 20| 20|
| 2| 40|null|
+---+---+---+

```

Please note that [Why do Window functions fail with "Window function X does not take a frame specification"?](#)

The key here is to remember that DataFrames are RDDs under the covers and hence aggregation like grouping by a key in DataFrames is RDD's `groupByKey` (or worse, `reduceByKey` or `aggregateByKey` transformations).

Running Total

The **running total** is the sum of all previous lines including the current one.

```
val sales = Seq(
  (0, 0, 0, 5),
  (1, 0, 1, 3),
  (2, 0, 2, 1),
  (3, 1, 0, 2),
  (4, 2, 0, 8),
  (5, 2, 2, 8))
  .toDF("id", "orderID", "prodID", "orderQty")

scala> sales.show
+---+-----+-----+-----+
| id|orderID|prodID|orderQty|
+---+-----+-----+-----+
| 0|      0|      0|      5|
| 1|      0|      1|      3|
| 2|      0|      2|      1|
| 3|      1|      0|      2|
| 4|      2|      0|      8|
| 5|      2|      2|      8|
+---+-----+-----+-----+

val orderedByID = Window.orderBy('id)

val totalQty = sum('orderQty).over(orderedByID).as('running_total)
val salesTotalQty = sales.select('*', totalQty).orderBy('id)

scala> salesTotalQty.show
16/04/10 23:01:52 WARN Window: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
+---+-----+-----+-----+
| id|orderID|prodID|orderQty|running_total|
+---+-----+-----+-----+
| 0|      0|      0|      5|      5|
| 1|      0|      1|      3|      8|
| 2|      0|      2|      1|      9|
| 3|      1|      0|      2|     11|
| 4|      2|      0|      8|     19|
| 5|      2|      2|      8|     27|
+---+-----+-----+-----+

val byOrderId = orderedByID.partitionBy('orderID)
val totalQtyPerOrder = sum('orderQty).over(byOrderId).as('running_total_per_order)
val salesTotalQtyPerOrder = sales.select('*', totalQtyPerOrder).orderBy('id)

scala> salesTotalQtyPerOrder.show
+---+-----+-----+-----+
| id|orderID|prodID|orderQty|running_total_per_order|
+---+-----+-----+-----+
```

	0	0	0	5	5
	1	0	1	3	8
	2	0	2	1	9
	3	1	0	2	2
	4	2	0	8	8
	5	2	2	8	16
+	-----+	-----+	-----+	-----+	

Calculate rank of row

See "[Explaining" Query Plans of Windows](#)" for an elaborate example.

Interval data type for Date and Timestamp types

See [\[SPARK-8943\] CalendarIntervalType for time intervals](#).

With the Interval data type, you could use intervals as values specified in `<value> PRECEDING` and `<value> FOLLOWING` for `RANGE` frame. It is specifically suited for time-series analysis with window functions.

Accessing values of earlier rows

FIXME What's the value of rows before current one?

Moving Average

Cumulative Aggregates

Eg. cumulative sum

User-defined aggregate functions

See [\[SPARK-3947\] Support Scala/Java UDAF](#).

With the window function support, you could use user-defined aggregate functions as window functions.

"Explaining" Query Plans of Windows

```

import org.apache.spark.sql.expressions.Window
val byDepnameSalaryDesc = Window.partitionBy('depname).orderBy('salary desc)

scala> val rankByDepname = rank().over(byDepnameSalaryDesc)
rankByDepname: org.apache.spark.sql.Column = RANK() OVER (PARTITION BY depname ORDER BY salary DESC UnspecifiedFrame)

// empsalary defined at the top of the page
scala> empsalary.select('*', rankByDepname as 'rank).explain(extended = true)
== Parsed Logical Plan ==
'Project [* , rank() windowspecdefinition('depname, 'salary DESC, UnspecifiedFrame) AS rank#9]
+- LocalRelation [depName#5, empNo#6L, salary#7L]

== Analyzed Logical Plan ==
depName: string, empNo: bigint, salary: bigint, rank: int
Project [depName#5, empNo#6L, salary#7L, rank#9]
+- Project [depName#5, empNo#6L, salary#7L, rank#9, rank#9]
  +- Window [rank(salary#7L) windowspecdefinition(depname#5, salary#7L DESC, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#9], [depname#5], [salary#7L DESC]
    +- Project [depName#5, empNo#6L, salary#7L]
    +- LocalRelation [depName#5, empNo#6L, salary#7L]

== Optimized Logical Plan ==
Window [rank(salary#7L) windowspecdefinition(depname#5, salary#7L DESC, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#9], [depname#5], [salary#7L DESC]
+- LocalRelation [depName#5, empNo#6L, salary#7L]

== Physical Plan ==
Window [rank(salary#7L) windowspecdefinition(depname#5, salary#7L DESC, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#9], [depname#5], [salary#7L DESC]
+- *Sort [depname#5 ASC, salary#7L DESC], false, 0
  +- Exchange hashpartitioning(depname#5, 200)
    +- LocalTableScan [depName#5, empNo#6L, salary#7L]

```

lag Window Function

```

lag(e: Column, offset: Int): Column
lag(columnName: String, offset: Int): Column
lag(columnName: String, offset: Int, defaultValue: Any): Column
lag(e: Column, offset: Int, defaultValue: Any): Column

```

`lag` returns the value in `e` / `columnName` column that is `offset` records before the current record. `lag` returns `null` value if the number of records in a window partition is less than `offset` or `defaultValue`.

```

val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("lag", lag('id, 1) over windowSpec).show
+---+-----+---+
| id|bucket| lag|
+---+-----+---+
| 0|     0| null|
| 3|     0|   0|
| 6|     0|   3|
| 1|     1| null|
| 4|     1|   1|
| 7|     1|   4|
| 2|     2| null|
| 5|     2|   2|
| 8|     2|   5|
+---+-----+---+

scala> dataset.withColumn("lag", lag('id, 2, "<default_value>") over windowSpec).show
+---+-----+---+
| id|bucket| lag|
+---+-----+---+
| 0|     0| null|
| 3|     0| null|
| 6|     0|   0|
| 1|     1| null|
| 4|     1| null|
| 7|     1|   1|
| 2|     2| null|
| 5|     2| null|
| 8|     2|   2|
+---+-----+---+

```

Caution

`FIXME` It looks like `lag` with a default value has a bug — the default value's not used at all.

lead Window Function

```

lead(columnName: String, offset: Int): Column
lead(e: Column, offset: Int): Column
lead(columnName: String, offset: Int, defaultValue: Any): Column
lead(e: Column, offset: Int, defaultValue: Any): Column

```

`lead` returns the value that is `offset` records after the current records, and `defaultValue` if there is less than `offset` records after the current record. `lag` returns `null` value if the number of records in a window partition is less than `offset` or `defaultValue`.

```
val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("lead", lead('id, 1) over windowSpec).show
+---+-----+-----+
| id|bucket|lead|
+---+-----+-----+
| 0|     0|    0|
| 0|     0|    3|
| 3|     0|    3|
| 3|     0|    6|
| 6|     0|    6|
| 6|     0|null|
| 1|     1|    1|
| 1|     1|    4|
| 4|     1|    4|
| 4|     1|    7|
| 7|     1|    7|
| 7|     1|null|
| 2|     2|    2|
| 2|     2|    5|
| 5|     2|    5|
| 5|     2|    8|
| 8|     2|    8|
| 8|     2|null|
+---+-----+-----+

scala> dataset.withColumn("lead", lead('id, 2, "<default_value>") over windowSpec).show
+---+-----+-----+
| id|bucket|lead|
+---+-----+-----+
| 0|     0|    3|
| 0|     0|    3|
| 3|     0|    6|
| 3|     0|    6|
| 6|     0|null|
| 6|     0|null|
| 1|     1|    4|
| 1|     1|    4|
| 4|     1|    7|
| 4|     1|    7|
| 7|     1|null|
| 7|     1|null|
| 2|     2|    5|
```

```
+---+-----+---+
| 2|     2|  5|
| 5|     2|  8|
| 5|     2|  8|
| 8|     2|null|
| 8|     2|null|
+---+-----+---+
```

Caution FIXME It looks like `lead` with a default value has a bug — the default value's not used at all.

Cumulative Distribution of Records Across Window Partitions — `cume_dist` Window Function

```
cume_dist(): Column
```

`cume_dist` computes the cumulative distribution of the records in window partitions. This is equivalent to SQL's `CUME_DIST` function.

```
val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("cume_dist", cume_dist over windowSpec).show
+---+-----+-----+
| id|bucket|      cume_dist|
+---+-----+-----+
|  0|     0|0|0.3333333333333333|
|  3|     0|0|0.6666666666666666|
|  6|     0|          1.0|
|  1|     1|1|0.3333333333333333|
|  4|     1|1|0.6666666666666666|
|  7|     1|          1.0|
|  2|     2|2|0.3333333333333333|
|  5|     2|2|0.6666666666666666|
|  8|     2|          1.0|
+---+-----+-----+
```

Sequential numbering per window partition — `row_number` Window Function

```
row_number(): Column
```

`row_number` returns a sequential number starting at `1` within a window partition.

```
val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("row_number", row_number() over windowSpec).show
+---+-----+
| id|bucket|row_number|
+---+-----+
|  0|     0|        1|
|  0|     0|        2|
|  3|     0|        3|
|  3|     0|        4|
|  6|     0|        5|
|  6|     0|        6|
|  1|     1|        1|
|  1|     1|        2|
|  4|     1|        3|
|  4|     1|        4|
|  7|     1|        5|
|  7|     1|        6|
|  2|     2|        1|
|  2|     2|        2|
|  5|     2|        3|
|  5|     2|        4|
|  8|     2|        5|
|  8|     2|        6|
+---+-----+
```

ntile Window Function

```
ntile(n: Int): Column
```

`ntile` computes the ntile group id (from `1` to `n` inclusive) in an ordered window partition.

```

val dataset = spark.range(7).select('*', 'id % 3 as "bucket")

import org.apache.spark.sql.expressions.Window
val byBuckets = Window.partitionBy('bucket).orderBy('id)
scala> dataset.select('*', ntile(3) over byBuckets as "ntile").show
+---+-----+
| id|bucket|ntile|
+---+-----+
|  0|      0|     1|
|  3|      0|     2|
|  6|      0|     3|
|  1|      1|     1|
|  4|      1|     2|
|  2|      2|     1|
|  5|      2|     2|
+---+-----+

```

Caution `FIXME` How is `ntile` different from `rank`? What about performance?

Ranking Records per Window Partition — `rank` Window Function

```

rank(): Column
dense_rank(): Column
percent_rank(): Column

```

`rank` functions assign the sequential rank of each distinct value per window partition. They are equivalent to `RANK`, `DENSE_RANK` and `PERCENT_RANK` functions in the good ol' SQL.

```

val dataset = spark.range(9).withColumn("bucket", 'id % 3)

import org.apache.spark.sql.expressions.Window
val byBucket = Window.partitionBy('bucket).orderBy('id)

scala> dataset.withColumn("rank", rank over byBucket).show
+---+-----+
| id|bucket|rank|
+---+-----+
|  0|     0|   1|
|  3|     0|   2|
|  6|     0|   3|
|  1|     1|   1|
|  4|     1|   2|
|  7|     1|   3|
|  2|     2|   1|
|  5|     2|   2|
|  8|     2|   3|
+---+-----+

scala> dataset.withColumn("percent_rank", percent_rank over byBucket).show
+---+-----+
| id|bucket|percent_rank|
+---+-----+
|  0|     0|      0.0|
|  3|     0|      0.5|
|  6|     0|      1.0|
|  1|     1|      0.0|
|  4|     1|      0.5|
|  7|     1|      1.0|
|  2|     2|      0.0|
|  5|     2|      0.5|
|  8|     2|      1.0|
+---+-----+

```

`rank` function assigns the same rank for duplicate rows with a gap in the sequence (similarly to Olympic medal places). `dense_rank` is like `rank` for duplicate rows but compacts the ranks and removes the gaps.

```

// rank function with duplicates
// Note the missing/sparse ranks, i.e. 2 and 4
scala> dataset.union(dataset).withColumn("rank", rank over byBucket).show
+---+-----+
| id|bucket|rank|
+---+-----+
|  0|     0|   1|
|  0|     0|   1|
|  3|     0|   3|
|  3|     0|   3|
|  6|     0|   5|
+---+-----+

```

```

| 6| 0| 5|
| 1| 1| 1|
| 1| 1| 1|
| 4| 1| 3|
| 4| 1| 3|
| 7| 1| 5|
| 7| 1| 5|
| 2| 2| 1|
| 2| 2| 1|
| 5| 2| 3|
| 5| 2| 3|
| 8| 2| 5|
| 8| 2| 5|
+---+---+---+
// dense_rank function with duplicates
// Note that the missing ranks are now filled in
scala> dataset.union(dataset).withColumn("dense_rank", dense_rank over byBucket).show
+---+---+---+
| id|bucket|dense_rank|
+---+---+---+
| 0| 0| 1|
| 0| 0| 1|
| 3| 0| 2|
| 3| 0| 2|
| 6| 0| 3|
| 6| 0| 3|
| 1| 1| 1|
| 1| 1| 1|
| 4| 1| 2|
| 4| 1| 2|
| 7| 1| 3|
| 7| 1| 3|
| 2| 2| 1|
| 2| 2| 1|
| 5| 2| 2|
| 5| 2| 2|
| 8| 2| 3|
| 8| 2| 3|
+---+---+---+
// percent_rank function with duplicates
scala> dataset.union(dataset).withColumn("percent_rank", percent_rank over byBucket).show
+---+---+---+
| id|bucket|percent_rank|
+---+---+---+
| 0| 0| 0.0|
| 0| 0| 0.0|
| 3| 0| 0.4|
| 3| 0| 0.4|
| 6| 0| 0.8|
| 6| 0| 0.8|

```

	1	1	0.0
	1	1	0.0
	4	1	0.4
	4	1	0.4
	7	1	0.8
	7	1	0.8
	2	2	0.0
	2	2	0.0
	5	2	0.4
	5	2	0.4
	8	2	0.8
	8	2	0.8

currentRow Window Function

```
currentRow(): Column
```

currentRow ...FIXME

unboundedFollowing Window Function

```
unboundedFollowing(): Column
```

unboundedFollowing ...FIXME

unboundedPreceding Window Function

```
unboundedPreceding(): Column
```

unboundedPreceding ...FIXME

Further Reading and Watching

- [Introducing Window Functions in Spark SQL](#)
- [3.5. Window Functions](#) in the official documentation of PostgreSQL
- [Window Functions in SQL](#)
- [Working with Window Functions in SQL Server](#)
- [OVER Clause \(Transact-SQL\)](#)

- An introduction to windowed functions
- Probably the Coolest SQL Feature: Window Functions
- Window Functions

UDFs — User-Defined Functions

User-Defined Functions (aka **UDF**) is a feature of Spark SQL to define new [Column-based](#) functions that extend the vocabulary of Spark SQL's DSL for transforming [Datasets](#).

<p>Important</p>	<p>Use the higher-level standard Column-based functions (with Dataset operators) whenever possible before reverting to developing user-defined functions since UDFs are a blackbox for Spark SQL and it cannot (and does not even try to) optimize them.</p> <p>As Reynold Xin from the Apache Spark project has once said on Spark's dev mailing list:</p> <p style="margin-left: 20px;">There are simple cases in which we can analyze the UDFs byte code and infer what it is doing, but it is pretty difficult to do in general.</p> <p>Check out UDFs are Blackbox — Don't Use Them Unless You've Got No Choice if you want to know the internals.</p>
-------------------------	---

You define a new UDF by defining a Scala function as an input parameter of `udf` function. It accepts Scala functions of up to 10 input parameters.

```

val dataset = Seq((0, "hello"), (1, "world")).toDF("id", "text")

// Define a regular Scala function
val upper: String => String = _.toUpperCase

// Define a UDF that wraps the upper Scala function defined above
// You could also define the function in place, i.e. inside udf
// but separating Scala functions from Spark SQL's UDFs allows for easier testing
import org.apache.spark.sql.functions.udf
val upperUDF = udf(upper)

// Apply the UDF to change the source dataset
scala> dataset.withColumn("upper", upperUDF('text)).show
+---+-----+
| id| text|upper|
+---+-----+
| 0|hello|HELLO|
| 1|world|WORLD|
+---+-----+

```

You can register UDFs to use in [SQL-based query expressions](#) via [UDFRegistration](#) (that is available through `SparkSession.udf` attribute).

```
val spark: SparkSession = ...
scala> spark.udf.register("myUpper", (input: String) => input.toUpperCase)
```

You can query for available [standard](#) and user-defined functions using the [Catalog](#) interface (that is available through `SparkSession.catalog` attribute).

```
val spark: SparkSession = ...
scala> spark.catalog.listFunctions.filter('name like "%upper%").show(false)
+-----+-----+-----+
|name |database|description|className |isTempor
ary|
+-----+-----+-----+
|myupper|null |null |null |true
 |
|upper |null |null |org.apache.spark.sql.catalyst.expressions.Upper|true
 |
+-----+-----+-----+
---+
```

Note

UDFs play a vital role in Spark MLlib to define new [Transformers](#) that are function objects that transform `DataFrames` into `DataFrames` by introducing new columns.

udf Functions (in functions object)

```
udf[RT: TypeTag](f: Function0[RT]): UserDefinedFunction
...
udf[RT: TypeTag, A1: TypeTag, A2: TypeTag, A3: TypeTag, A4: TypeTag, A5: TypeTag, A6: TypeTag, A7: TypeTag, A8: TypeTag, A9: TypeTag, A10: TypeTag](f: Function10[A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, RT]): UserDefinedFunction
```

`org.apache.spark.sql.functions` object comes with `udf` function to let you define a UDF for a Scala function `f`.

```
val df = Seq(
  (0, "hello"),
  (1, "world")).toDF("id", "text")

// Define a "regular" Scala function
// It's a clone of upper UDF
val toUpper: String => String = _.toUpperCase

import org.apache.spark.sql.functions.udf
val upper = udf(toUpper)

scala> df.withColumn("upper", upper('text)).show
+---+-----+
| id| text|upper|
+---+-----+
|  0|hello|HELLO|
|  1|world|WORLD|
+---+-----+

// You could have also defined the UDF this way
val upperUDF = udf { s: String => s.toUpperCase }

// or even this way
val upperUDF = udf[String, String](_.toUpperCase)

scala> df.withColumn("upper", upperUDF('text)).show
+---+-----+
| id| text|upper|
+---+-----+
|  0|hello|HELLO|
|  1|world|WORLD|
+---+-----+
```

Tip

Define custom UDFs based on "standalone" Scala functions (e.g. `toUpperUDF`) so you can test the Scala functions using Scala way (without Spark SQL's "noise") and once they are defined reuse the UDFs in [UnaryTransformers](#).

UDFs are Blackbox — Don't Use Them Unless You've Got No Choice

Let's review an example with an UDF. This example is converting strings of size 7 characters only and uses the `Dataset` standard operators first and then custom UDF to do the same transformation.

```
scala> spark.conf.get("spark.sql.parquet.filterPushdown")
res0: String = true
```

You are going to use the following `cities` dataset that is based on Parquet file (as used in [Predicate Pushdown / Filter Pushdown for Parquet Data Source](#) section). The reason for parquet is that it is an external data source that does support optimization Spark uses to optimize itself like predicate pushdown.

```
// no optimization as it is a more involved Scala function in filter
// 08/30 Asked on dev@spark mailing list for explanation
val cities6chars = cities.filter(_.name.length == 6).map(_.name.toUpperCase)

cities6chars.explain(true)

// or simpler when only concerned with PushedFilters attribute in Parquet
scala> cities6chars.queryExecution.optimizedPlan
res33: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#248]
+- MapElements <function1>, class City, [StructField(id,LongType,false), StructField(name, StringType, true)], obj#247: java.lang.String
    +- Filter <function1>.apply
        +- DeserializeToObject newInstance(class City), obj#246: City
            +- Relation[id#236L, name#237] parquet

// no optimization for Dataset[City]!?
// 08/30 Asked on dev@spark mailing list for explanation
val cities6chars = cities.filter(_.name == "Warsaw").map(_.name.toUpperCase)

cities6chars.explain(true)

// The filter predicate is pushed down fine for Dataset's Column-based query in where
operator
scala> cities.where('name === "Warsaw").queryExecution.executedPlan
res29: org.apache.spark.sql.execution.SparkPlan =
*Project [id#128L, name#129]
+- *Filter (isnotnull(name#129) && (name#129 = Warsaw))
   +- *FileScan parquet [id#128L, name#129] Batched: true, Format: ParquetFormat, Input
Paths: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFil
ters: [IsNotNull(name), EqualTo(name, Warsaw)], ReadSchema: struct<id:bigint, name:strin
g>

// Let's define a UDF to do the filtering
val isWarsaw = udf { (s: String) => s == "Warsaw" }

// Use the UDF in where (replacing the Column-based query)
scala> cities.where(isWarsaw('name)).queryExecution.executedPlan
res33: org.apache.spark.sql.execution.SparkPlan =
*Filter UDF(name#129)
+- *FileScan parquet [id#128L, name#129] Batched: true, Format: ParquetFormat, InputPat
hs: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFilters
: [], ReadSchema: struct<id:bigint, name:string>
```

UserDefinedFunction

UserDefinedFunction represents a **user-defined function**.

UserDefinedFunction is created when:

- udf function is executed
- UDFRegistration is requested to register a Scala function as a user-defined function (in FunctionRegistry)

```
import org.apache.spark.sql.functions.udf
val lengthUDF = udf { s: String => s.length }

scala> :type lengthUDF
org.apache.spark.sql.expressions.UserDefinedFunction

val r = lengthUDF($"name")

scala> :type r
org.apache.spark.sql.Column
```

UserDefinedFunction can have an optional name.

```
val namedLengthUDF = lengthUDF.withName("lengthUDF")
scala> namedLengthUDF($"name")
res2: org.apache.spark.sql.Column = UDF:lengthUDF(name)
```

UserDefinedFunction is **nullable** by default, but can be changed as **non-nullable**.

```
val nonNullableLengthUDF = lengthUDF.asNonNullable
assert(nonNullableLengthUDF.nullable == false)
```

UserDefinedFunction is **deterministic** by default, i.e. produces the same result for the same input. UserDefinedFunction can be changed to be **non-deterministic**.

```
assert(lengthUDF.deterministic)
val ndUDF = lengthUDF.asNondeterministic
assert(ndUDF.deterministic == false)
```

Table 1. UserDefinedFunction's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
_deterministic	<p>Flag that controls whether the function is deterministic (<code>true</code>) or not (<code>false</code>).</p> <p>Default: <code>true</code></p> <ul style="list-style-type: none"> • Use asNondeterministic to change it to <code>false</code> <p>Used when <code>UserDefinedFunction</code> is requested to execute</p>

Executing UserDefinedFunction (Creating Column with ScalaUDF Expression) — `apply` Method

```
apply(exprs: Column*): Column
```

`apply` creates a [Column](#) with [ScalaUDF](#) expression.

```
import org.apache.spark.sql.functions.udf
scala> val lengthUDF = udf { s: String => s.length }
lengthUDF: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(
<function1>, IntegerType, Some(List(StringType)))

scala> lengthUDF($"name")
res1: org.apache.spark.sql.Column = UDF(name)
```

Note	<code>apply</code> is used when...FIXME
------	---

Marking UserDefinedFunction as NonNullable — `asNonNullable` Method

```
asNonNullable(): UserDefinedFunction
```

`asNonNullable` ...FIXME

Note	<code>asNonNullable</code> is used when...FIXME
------	---

Naming UserDefinedFunction — `withName` Method

```
withName(name: String): UserDefinedFunction
```

`withName ...FIXME`

Note

`withName` is used when...FIXME

Creating UserDefinedFunction Instance

`UserDefinedFunction` takes the following when created:

- A Scala function (as Scala's `AnyRef`)
- Output [data type](#)
- Input [data types](#) (if available)

`UserDefinedFunction` initializes the [internal registries and counters](#).

Schema — Structure of Data

A **schema** is the description of the structure of your data (which together create a [Dataset](#) in Spark SQL). It can be **implicit** (and [inferred at runtime](#)) or **explicit** (and known at compile time).

A schema is described using [StructType](#) which is a collection of [StructField](#) objects (that in turn are tuples of names, types, and `nullability` classifier).

`StructType` and `StructField` belong to the `org.apache.spark.sql.types` package.

```
import org.apache.spark.sql.types.StructType
val schemaUntyped = new StructType()
  .add("a", "int")
  .add("b", "string")

// alternatively using Schema DSL
val schemaUntyped_2 = new StructType()
  .add($"a".int)
  .add($"b".string)
```

You can use the canonical string representation of SQL types to describe the types in a schema (that is inherently untyped at compile type) or use type-safe types from the `org.apache.spark.sql.types` package.

```
// it is equivalent to the above expressions
import org.apache.spark.sql.types.{IntegerType, StringType}
val schemaTyped = new StructType()
  .add("a", IntegerType)
  .add("b", StringType)
```

Tip	Read up on CatalystSqlParser that is responsible for parsing data types.
-----	--

It is however recommended to use the singleton [DataTypes](#) class with static methods to create schema types.

```
import org.apache.spark.sql.types.DataTypes._
val schemaWithMap = StructType(
  StructField("map", createMapType(LongType, StringType), false) :: Nil)
```

`StructType` offers [printTreeString](#) that makes presenting the schema more user-friendly.

```

scala> schemaTyped.printTreeString
root
|-- a: integer (nullable = true)
|-- b: string (nullable = true)

scala> schemaWithMap.printTreeString
root
|-- map: map (nullable = false)
|   |-- key: long
|   |-- value: string (valueContainsNull = true)

// You can use prettyJson method on any DataType
scala> println(schema1.prettyJson)
{
  "type" : "struct",
  "fields" : [ {
    "name" : "a",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "b",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  } ]
}

```

As of Spark 2.0, you can describe the schema of your strongly-typed datasets using [encoders](#).

```

import org.apache.spark.sql.Encoders

scala> Encoders.INT.schema.printTreeString
root
|-- value: integer (nullable = true)

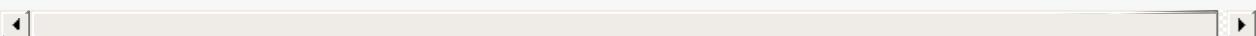
scala> Encoders.product[(String, java.sql.Timestamp)].schema.printTreeString
root
|-- _1: string (nullable = true)
|-- _2: timestamp (nullable = true)

case class Person(id: Long, name: String)
scala> Encoders.product[Person].schema.printTreeString
root
|-- id: long (nullable = false)
|-- name: string (nullable = true)

```

Implicit Schema

```
val df = Seq((0, s"""hello\tworld"""), (1, "two  spaces inside")).toDF("label", "sentence")  
  
scala> df.printSchema  
root  
| -- label: integer (nullable = false)  
| -- sentence: string (nullable = true)  
  
scala> df.schema  
res0: org.apache.spark.sql.types.StructType = StructType(StructField(label,IntegerType,  
false), StructField(sentence,StringType,true))  
  
scala> df.schema("label").dataType  
res1: org.apache.spark.sql.types.DataType = IntegerType
```



StructType — Data Type for Schema Definition

`StructType` is a built-in [data type](#) that is a collection of [StructFields](#).

`StructType` is used to define a schema or its part.

You can compare two `StructType` instances to see whether they are equal.

```
import org.apache.spark.sql.types.StructType

val schemaUntyped = new StructType()
  .add("a", "int")
  .add("b", "string")

import org.apache.spark.sql.types.{IntegerType, StringType}
val schemaTyped = new StructType()
  .add("a", IntegerType)
  .add("b", StringType)

scala> schemaUntyped == schemaTyped
res0: Boolean = true
```

`StructType` presents itself as `<struct>` or `STRUCT` in query plans or SQL.

Note

`StructType` is a `Seq[StructField]` and therefore all things `seq` apply equally here.

```
scala> schemaTyped.foreach(println)
StructField(a,IntegerType,true)
StructField(b,StringType,true)
```

Read the official documentation of Scala's [scala.collection.Seq](#).

As of Spark 2.4.0, `structType` can be converted to DDL format using `toDDL` method.

Example: Using `StructType.toDDL`

```
// Generating a schema from a case class
// Because we're all properly lazy
case class Person(id: Long, name: String)
import org.apache.spark.sql.Encoders
val schema = Encoders.product[Person].schema
scala> println(schema.toDDL)
`id` BIGINT, `name` STRING
```

fromAttributes Method

```
fromAttributes(attributes: Seq[Attribute]): StructType
```

fromAttributes ...FIXME

Note

fromAttributes is used when...FIXME

toAttributes Method

```
toAttributes: Seq[AttributeReference]
```

toAttributes ...FIXME

Note

toAttributes is used when...FIXME

Adding Fields to Schema — add Method

You can add a new `StructField` to your `StructType`. There are different variants of `add` method that all make for a new `StructType` with the field added.

```

add(field: StructField): StructType
add(name: String, dataType: DataType): StructType
add(name: String, dataType: DataType, nullable: Boolean): StructType
add(
  name: String,
  dataType: DataType,
  nullable: Boolean,
  metadata: Metadata): StructType
add(
  name: String,
  dataType: DataType,
  nullable: Boolean,
  comment: String): StructType
add(name: String, dataType: String): StructType
add(name: String, dataType: String, nullable: Boolean): StructType
add(
  name: String,
  dataType: String,
  nullable: Boolean,
  metadata: Metadata): StructType
add(
  name: String,
  dataType: String,
  nullable: Boolean,
  comment: String): StructType

```

Data Type Name Conversions

```

simpleString: String
catalogString: String
sql: String

```

`StructType` as a custom `DataType` is used in query plans or SQL. It can present itself using `simpleString`, `catalogString` or `sql` (see [DataType Contract](#)).

```

scala> schemaTyped.simpleString
res0: String = struct<a:int,b:string>

scala> schemaTyped.catalogString
res1: String = struct<a:int,b:string>

scala> schemaTyped.sql
res2: String = STRUCT<`a`: INT, `b`: STRING>

```

Accessing StructField — apply Method

```
apply(name: String): StructField
```

`StructType` defines its own `apply` method that gives you an easy access to a `StructField` by name.

```
scala> schemaTyped.printTreeString
root
|-- a: integer (nullable = true)
|-- b: string (nullable = true)

scala> schemaTyped("a")
res4: org.apache.spark.sql.types.StructField = StructField(a, IntegerType, true)
```

Creating StructType from Existing StructType — `apply` Method

```
apply(names: Set[String]): StructType
```

This variant of `apply` lets you create a `StructType` out of an existing `StructType` with the `names` only.

```
scala> schemaTyped(names = Set("a"))
res0: org.apache.spark.sql.types.StructType = StructType(StructField(a, IntegerType, true
))
```

It will throw an `IllegalArgumentException` exception when a field could not be found.

```
scala> schemaTyped(names = Set("a", "c"))
java.lang.IllegalArgumentException: Field c does not exist.
  at org.apache.spark.sql.types.StructType.apply(StructType.scala:275)
... 48 elided
```

Displaying Schema As Tree — `printTreeString` Method

```
printTreeString(): Unit
```

`printTreeString` prints out the schema to standard output.

```
scala> schemaTyped.printTreeString
root
|-- a: integer (nullable = true)
|-- b: string (nullable = true)
```

Internally, it uses `treeString` method to build the tree and then `println` it.

Creating StructType For DDL-F\$formatted Text — `fromDDL` Object Method

```
fromDDL(ddl: String): StructType
```

`fromDDL ...FIXME`

Note	<code>fromDDL</code> is used when...FIXME
------	---

Converting to DDL Format — `toDDL` Method

```
toDDL: String
```

`toDDL` converts all the `fields` to DDL format and concatenates them using the comma (,).

StructField — Single Field in StructType

`StructField` describes a single field in a `StructType` with the following:

- Name
- `DataType`
- `nullable` flag (enabled by default)
- `Metadata` (empty by default)

A comment is part of metadata under `comment` key and is used to build a Hive column or when describing a table.

```
scala> schemaTyped("a").getComment
res0: Option[String] = None

scala> schemaTyped("a").withComment("this is a comment").getComment
res1: Option[String] = Some(this is a comment)
```

As of Spark 2.4.0, `structField` can be converted to DDL format using `toDDL` method.

Example: Using `StructField.toDDL`

```
import org.apache.spark.sql.types.MetadataBuilder
val metadata = new MetadataBuilder()
  .putString("comment", "this is a comment")
  .build
import org.apache.spark.sql.types.{LongType, StructField}
val f = new StructField(name = "id", dataType = LongType, nullable = false, metadata)
scala> println(f.toDDL)
`id` BIGINT COMMENT 'this is a comment'
```

Converting to DDL Format — `toDDL` Method

```
toDDL: String
```

`toDDL` gives a text in the format:

```
[quoted name] [dataType][optional comment]
```

	<p><code>toDDL</code> is used when:</p> <ul style="list-style-type: none">• <code>StructType</code> is requested to convert itself to DDL format• <code>ShowCreateTableCommand</code> logical command is executed (and <code>showHiveTableHeader</code>, <code>showHiveTableNonDataColumns</code>, <code>showDataSourceTableDataColumns</code>)
Note	

Data Types

`DataType` abstract class is the base type of all built-in data types in Spark SQL, e.g. strings, longs.

`DataType` has two main type families:

- **Atomic Types** as an internal type to represent types that are not `null`, UDTs, arrays, structs, and maps
- **Numeric Types** with `fractional` and `integral` types

Table 1. Standard Data Types

Type Family	Data Type	Scala Types
Atomic Types <small>(except fractional and integral types)</small>	BinaryType	
	BooleanType	
	DateType	
	StringType	
	TimestampType	java.sql.Timestamp
Fractional Types <small>(concrete NumericType)</small>	DecimalType	
	DoubleType	
	FloatType	
Integral Types <small>(concrete NumericType)</small>	ByteType	
	IntegerType	
	LongType	
	ShortType	
	ArrayType	
	CalendarIntervalType	
	MapType	
	NullType	
	ObjectType	
	StructType	
	UserDefinedType	
	AnyDataType	Matches any concrete data type

Caution

FIXME What about AbstractDataType?

You can extend the type system and create your own [user-defined types \(UDTs\)](#).

The [DataType Contract](#) defines methods to build SQL, JSON and string representations.

Note

`DataType` (and the concrete Spark SQL types) live in `org.apache.spark.sql.types` package.

```
import org.apache.spark.sql.types.StringType

scala> StringType.json
res0: String = "string"

scala> StringType.sql
res1: String = STRING

scala> StringType.catalogString
res2: String = string
```

You should use `DataTypes` object in your code to create complex Spark SQL types, i.e. arrays or maps.

```
import org.apache.spark.sql.types.DataTypes

scala> val arrayType = DataTypes.createArrayType(BooleanType)
arrayType: org.apache.spark.sql.types.ArrayType = ArrayType(BooleanType, true)

scala> val mapType = DataTypes.createMapType(StringType, LongType)
mapType: org.apache.spark.sql.types.MapType = MapType(StringType, LongType, true)
```

`DataType` has support for Scala's pattern matching using `unapply` method.

```
???
```

DataType Contract

Any type in Spark SQL follows the `DataType` contract which means that the types define the following methods:

- `json` and `prettyJson` to build JSON representations of a data type
- `defaultSize` to know the default size of values of a type
- `simpleString` and `catalogString` to build user-friendly string representations (with the latter for external catalogs)
- `sql` to build SQL representation

```

import org.apache.spark.sql.types.DataTypes._

val maps = StructType(
  StructField("longs2strings", createMapType(LongType, StringType), false) :: Nil)

scala> maps.prettyJson
res0: String =
{
  "type" : "struct",
  "fields" : [ {
    "name" : "longs2strings",
    "type" : {
      "type" : "map",
      "keyType" : "long",
      "valueType" : "string",
      "valueContainsNull" : true
    },
    "nullable" : false,
    "metadata" : { }
  } ]
}

scala> maps.defaultSize
res1: Int = 2800

scala> maps.simpleString
res2: String = struct<longs2strings:map<bigint,string>>

scala> maps.catalogString
res3: String = struct<longs2strings:map<bigint,string>>

scala> maps.sql
res4: String = STRUCT<`longs2strings`: MAP<BIGINT, STRING>>

```

DataTypes — Factory Methods for Data Types

`DataTypes` is a Java class with methods to access simple or create complex `DataType` types in Spark SQL, i.e. arrays and maps.

Tip

It is recommended to use `DataTypes` class to define `DataType` types in a schema.

`DataTypes` lives in `org.apache.spark.sql.types` package.

```
import org.apache.spark.sql.types.DataTypes

scala> val arrayType = DataTypes.createArrayType(BooleanType)
arrayType: org.apache.spark.sql.types.ArrayType = ArrayType(BooleanType, true)

scala> val mapType = DataTypes.createMapType(StringType, LongType)
mapType: org.apache.spark.sql.types.MapType = MapType(StringType, LongType, true)
```

Note

Simple `DataType` types themselves, i.e. `StringType` or `CalendarIntervalType`, come with their own Scala's `case object`s alongside their definitions.

You may also import the `types` package and have access to the types.

```
import org.apache.spark.sql.types._
```

UDTs — User-Defined Types

Caution

FIXME

Converting DDL into DataType — `fromDDL` Utility

```
fromDDL(ddl: String): DataType
```

```
fromDDL ...FIXME
```

Note

`fromDDL` is used when...FIXME

Multi-Dimensional Aggregation

Multi-dimensional aggregate operators are enhanced variants of `groupBy` operator that allow you to create queries for subtotals, grand totals and superset of subtotals in one go.

```
val sales = Seq(
    ("Warsaw", 2016, 100),
    ("Warsaw", 2017, 200),
    ("Boston", 2015, 50),
    ("Boston", 2016, 150),
    ("Toronto", 2017, 50)
).toDF("city", "year", "amount")

// very labor-intense
// groupBy's unioned
val groupByCityAndYear = sales
    .groupBy("city", "year") // <-- subtotals (city, year)
    .agg(sum("amount") as "amount")
val groupByCityOnly = sales
    .groupBy("city")         // <-- subtotals (city)
    .agg(sum("amount") as "amount")
    .select($"city", lit(null) as "year", $"amount") // <-- year is null
val withUnion = groupByCityAndYear
    .union(groupByCityOnly)
    .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> withUnion.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
+-----+-----+
```

Multi-dimensional aggregate operators are semantically equivalent to `union` operator (or SQL's `UNION ALL`) to combine single grouping queries.

```
// Roll up your sleeves!
val withRollup = sales
  .rollup("city", "year")
  .agg(sum("amount") as "amount", grouping_id() as "gid")
  .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
  .filter(grouping_id() != 3)
  .select("city", "year", "amount")
scala> withRollup.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|   300|
| Toronto|2017|     50|
| Toronto|null|    50|
| Boston|2015|     50|
| Boston|2016|   150|
| Boston|null|   200|
+-----+-----+

// Be even more smarter?
// SQL only, alas.
sales.createOrReplaceTempView("sales")
val withGroupingSets = sql("""
  SELECT city, year, SUM(amount) as amount
  FROM sales
  GROUP BY city, year
  GROUPING SETS ((city, year), (city))
  ORDER BY city DESC NULLS LAST, year ASC NULLS LAST
  """)

scala> withGroupingSets.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|   300|
| Toronto|2017|     50|
| Toronto|null|    50|
| Boston|2015|     50|
| Boston|2016|   150|
| Boston|null|   200|
+-----+-----+
```

Note

It is *assumed* that using one of the operators is usually more efficient (than `union` and `groupBy`) as it gives more freedom for query optimization.

Table 1. Multi-dimensional Aggregate Operators

Operator	Return Type	Description
<code>cube</code>	<code>RelationalGroupedDataset</code>	Calculates subtotals and a grand total for every permutation of the columns specified.
<code>rollup</code>	<code>RelationalGroupedDataset</code>	Calculates subtotals and a grand total over (ordered) combination of groups.

Beside `cube` and `rollup` multi-dimensional aggregate operators, Spark SQL supports `GROUPING SETS` clause in SQL mode only.

Note	SQL's <code>GROUPING SETS</code> is the most general aggregate "operator" and can generate the same dataset as using a simple <code>groupBy</code> , <code>cube</code> and <code>rollup</code> operators.
------	---

```

import java.time.LocalDate
import java.sql.Date
val expenses = Seq(
  ((2012, Month.DECEMBER, 12), 5),
  ((2016, Month.AUGUST, 13), 10),
  ((2017, Month.MAY, 27), 15))
  .map { case ((yy, mm, dd), a) => (LocalDate.of(yy, mm, dd), a) }
  .map { case (d, a) => (d.toString, a) }
  .map { case (d, a) => (Date.valueOf(d), a) }
  .toDF("date", "amount")
scala> expenses.show
+-----+-----+
|      date|amount|
+-----+-----+
|2012-12-12|     5|
|2016-08-13|    10|
|2017-05-27|    15|
+-----+-----+

// rollup time!
val q = expenses
  .rollup(year($"date") as "year", month($"date") as "month")
  .agg(sum("amount") as "amount")
  .sort($"year".asc_nulls_last, $"month".asc_nulls_last)
scala> q.show
+----+----+-----+
|year|month|amount|
+----+----+-----+
|2012|   12|     5|
|2012| null|     5|
|2016|    8|    10|
|2016| null|    10|
|2017|    5|    15|
|2017| null|    15|
|null| null|    30|
+----+----+-----+

```

Tip Review the examples per operator in the following sections.

Note Support for multi-dimensional aggregate operators was added in [\[SPARK-6356\]](#)
[Support the ROLLUP/CUBE/GROUPING SETS/grouping\(\) in SQLContext.](#)

rollup Operator

```

rollup(cols: Column*): RelationalGroupedDataset
rollup(col1: String, cols: String*): RelationalGroupedDataset

```

`rollup` multi-dimensional aggregate operator is an extension of `groupBy` operator that calculates subtotals and a grand total across specified group of `n + 1` dimensions (with `n` being the number of columns as `cols` and `col1` and `1` for where values become `null`, i.e. undefined).

Note

`rollup` operator is commonly used for analysis over hierarchical data; e.g. total salary by department, division, and company-wide total.

See PostgreSQL's [7.2.4. GROUPING SETS, CUBE, and ROLLUP](#)

Note

`rollup` operator is equivalent to `GROUP BY ... WITH ROLLUP` in SQL (which in turn is equivalent to `GROUP BY ... GROUPING SETS ((a,b,c),(a,b),(a),())` when used with 3 columns: `a`, `b`, and `c`).

```
val sales = Seq(
  ("Warsaw", 2016, 100),
  ("Warsaw", 2017, 200),
  ("Boston", 2015, 50),
  ("Boston", 2016, 150),
  ("Toronto", 2017, 50)
).toDF("city", "year", "amount")

val q = sales
  .rollup("city", "year")
  .agg(sum("amount") as "amount")
  .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> q.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100| <- subtotal for Warsaw in 2016
| Warsaw|2017|    200|
| Warsaw|null|    300| <- subtotal for Warsaw (across years)
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
|   null|null|    550| <- grand total
+-----+-----+

// The above query is semantically equivalent to the following
val q1 = sales
  .groupBy("city", "year") // <- subtotals (city, year)
  .agg(sum("amount") as "amount")
val q2 = sales
  .groupBy("city")        // <- subtotals (city)
  .agg(sum("amount") as "amount")
  .select($"city", lit(null) as "year", $"amount") // <- year is null
val q3 = sales
```

```
.groupBy()           // <-- grand total
.agg(sum("amount") as "amount")
.select(lit(null) as "city", lit(null) as "year", $"amount") // <-- city and year are null
val qq = q1
.union(q2)
.union(q3)
.sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> qq.show
+-----+---+-----+
|   city|year|amount|
+-----+---+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
|    null|null|    550|
+-----+---+-----+
```

From [Using GROUP BY with ROLLUP, CUBE, and GROUPING SETS in Microsoft's TechNet](#):

The ROLLUP, CUBE, and GROUPING SETS operators are extensions of the GROUP BY clause. The ROLLUP, CUBE, or GROUPING SETS operators can generate the same result set as when you use UNION ALL to combine single grouping queries; however, using one of the GROUP BY operators is usually more efficient.

From [PostgreSQL's 7.2.4. GROUPING SETS, CUBE, and ROLLUP](#):

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear.

From [Summarizing Data Using ROLLUP in Microsoft's TechNet](#):

The ROLLUP operator is useful in generating reports that contain subtotals and totals. (...) ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

```
// Borrowed from Microsoft's "Summarizing Data Using ROLLUP" article
val inventory = Seq(
  ("table", "blue", 124),
  ("table", "red", 223),
  ("chair", "blue", 101),
  ("chair", "red", 210)).toDF("item", "color", "quantity")

scala> inventory.show
+---+---+-----+
| item|color|quantity|
+---+---+-----+
|chair| blue|    101|
|chair| red|    210|
|table| blue|    124|
|table| red|    223|
+---+---+-----+

// ordering and empty rows done manually for demo purposes
scala> inventory.rollup("item", "color").sum().show
+---+---+-----+
| item|color|sum(quantity)|
+---+---+-----+
|chair| blue|      101|
|chair| red|      210|
|chair| null|      311|
|   |   |      |
|table| blue|      124|
|table| red|      223|
|table| null|      347|
|   |   |      |
| null| null|      658|
+---+---+-----+
```

From Hive's Cubes and Rollups:

WITH ROLLUP is used with the GROUP BY only. ROLLUP clause is used with GROUP BY to compute the aggregate at the hierarchy levels of a dimension.

GROUP BY a, b, c with ROLLUP assumes that the hierarchy is "a" drilling down to "b" drilling down to "c".

GROUP BY a, b, c, WITH ROLLUP is equivalent to GROUP BY a, b, c GROUPING SETS ((a, b, c), (a, b), (a), ()).

Note	Read up on ROLLUP in Hive's LanguageManual in Grouping Sets, Cubes, Rollups, and the GROUPING_ID Function .
------	---

```
// Borrowed from http://stackoverflow.com/a/27222655/1305344
val quarterlyScores = Seq(
  ("winter2014", "Agata", 99),
  ("winter2014", "Jacek", 97),
  ("summer2015", "Agata", 100),
  ("summer2015", "Jacek", 63),
  ("winter2015", "Agata", 97),
  ("winter2015", "Jacek", 55),
  ("summer2016", "Agata", 98),
  ("summer2016", "Jacek", 97)).toDF("period", "student", "score")

scala> quarterlyScores.show
+-----+-----+-----+
| period|student|score|
+-----+-----+-----+
|winter2014|  Agata|   99|
|winter2014|  Jacek|   97|
|summer2015|  Agata|  100|
|summer2015|  Jacek|   63|
|winter2015|  Agata|   97|
|winter2015|  Jacek|   55|
|summer2016|  Agata|   98|
|summer2016|  Jacek|   97|
+-----+-----+-----+

// ordering and empty rows done manually for demo purposes
scala> quarterlyScores.rollup("period", "student").sum("score").show
+-----+-----+-----+
| period|student|sum(score)|
+-----+-----+-----+
|winter2014|  Agata|     99|
|winter2014|  Jacek|     97|
|winter2014|    null|    196|
|           |        |        |
|summer2015|  Agata|    100|
|summer2015|  Jacek|    63|
|summer2015|    null|    163|
|           |        |        |
|winter2015|  Agata|    97|
|winter2015|  Jacek|    55|
|winter2015|    null|    152|
|           |        |        |
|summer2016|  Agata|    98|
|summer2016|  Jacek|    97|
|summer2016|    null|    195|
|           |        |        |
|    null|    null|    706|
+-----+-----+-----+
```

From PostgreSQL's 7.2.4. GROUPING SETS, CUBE, and ROLLUP:

The individual elements of a CUBE or ROLLUP clause may be either individual expressions, or sublists of elements in parentheses. In the latter case, the sublists are treated as single units for the purposes of generating the individual grouping sets.

```
// given the above inventory dataset

// using struct function
scala> inventory.rollup(struct("item", "color") as "(item,color)").sum().show
+-----+-----+
|(item,color)|sum(quantity)|
+-----+-----+
| [table,red]|      223|
| [chair,blue]|     101|
|      null|     658|
| [chair,red]|     210|
| [table,blue]|     124|
+-----+-----+

// using expr function
scala> inventory.rollup(expr("(item, color)") as "(item, color)").sum().show
+-----+-----+
|(item, color)|sum(quantity)|
+-----+-----+
| [table,red]|      223|
| [chair,blue]|     101|
|      null|     658|
| [chair,red]|     210|
| [table,blue]|     124|
+-----+-----+
```

Internally, `rollup` converts the `Dataset` into a `DataFrame` (i.e. uses `RowEncoder` as the encoder) and then creates a `RelationalGroupedDataset` (with `RollupType` group type).

Note

`Rollup` expression represents `GROUP BY ... WITH ROLLUP` in SQL in Spark's Catalyst Expression tree (after `AstBuilder` parses a structured query with aggregation).

Tip

Read up on `rollup` in [Deeper into Postgres 9.5 - New Group By Options for Aggregation](#).

cube Operator

```
cube(cols: Column*): RelationalGroupedDataset
cube(col1: String, cols: String*): RelationalGroupedDataset
```

`cube` multi-dimensional aggregate operator is an extension of [groupBy](#) operator that allows calculating subtotals and a grand total across all combinations of specified group of `n + 1` dimensions (with `n` being the number of columns as `cols` and `col1` and `1` for where values become `null`, i.e. undefined).

`cube` returns [RelationalGroupedDataset](#) that you can use to execute aggregate function or operator.

Note

`cube` is more than [rollup](#) operator, i.e. `cube` does [rollup](#) with aggregation over all the missing combinations given the columns.

```
val sales = Seq(
  ("Warsaw", 2016, 100),
  ("Warsaw", 2017, 200),
  ("Boston", 2015, 50),
  ("Boston", 2016, 150),
  ("Toronto", 2017, 50)
).toDF("city", "year", "amount")

val q = sales.cube("city", "year")
  .agg(sum("amount") as "amount")
  .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> q.show
+-----+----+
| city|year|amount|
+-----+----+
| Warsaw|2016|   100| <-- total in Warsaw in 2016
| Warsaw|2017|   200| <-- total in Warsaw in 2017
| Warsaw|null|   300| <-- total in Warsaw (across all years)
| Toronto|2017|    50|
| Toronto|null|    50|
| Boston|2015|    50|
| Boston|2016|   150|
| Boston|null|   200|
|   null|2015|    50| <-- total in 2015 (across all cities)
|   null|2016|   250|
|   null|2017|   250|
|   null|null|   550| <-- grand total (across cities and years)
+-----+----+
```

GROUPING SETS SQL Clause

```
GROUP BY ... GROUPING SETS (...)
```

`GROUPING SETS` clause generates a dataset that is equivalent to `union` operator of multiple [groupBy](#) operators.

```

val sales = Seq(
  ("Warsaw", 2016, 100),
  ("Warsaw", 2017, 200),
  ("Boston", 2015, 50),
  ("Boston", 2016, 150),
  ("Toronto", 2017, 50)
).toDF("city", "year", "amount")
sales.createOrReplaceTempView("sales")

// equivalent to rollup("city", "year")
val q = sql("""
  SELECT city, year, sum(amount) as amount
  FROM sales
  GROUP BY city, year
  GROUPING SETS ((city, year), (city), ())
  ORDER BY city DESC NULLS LAST, year ASC NULLS LAST
""")

scala> q.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
| null|null|   550| <- grand total across all cities and years
+-----+-----+

// equivalent to cube("city", "year")
// note the additional (year) grouping set
val q = sql("""
  SELECT city, year, sum(amount) as amount
  FROM sales
  GROUP BY city, year
  GROUPING SETS ((city, year), (city), (year), ())
  ORDER BY city DESC NULLS LAST, year ASC NULLS LAST
""")

scala> q.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|

```

```
| Boston|null| 200|
| null|2015| 50| <-- total across all cities in 2015
| null|2016| 250| <-- total across all cities in 2016
| null|2017| 250| <-- total across all cities in 2017
| null|null| 550|
+-----+-----+
```

Internally, `GROUPING SETS` clause is parsed in [withAggregation](#) parsing handler (in `AstBuilder`) and becomes a [GroupingSets](#) logical operator internally.

Rollup GroupingSet with CodegenFallback Expression (for rollup Operator)

```
Rollup(groupByExprs: Seq[Expression])
extends GroupingSet
```

`Rollup` expression represents `rollup` operator in Spark's Catalyst Expression tree (after `AstBuilder` parses a structured query with aggregation).

Note	<code>GroupingSet</code> is an Expression with CodegenFallback support.
------	---

Dataset Caching and Persistence

One of the optimizations in Spark SQL is **Dataset caching** (aka **Dataset persistence**) which is available using the [Dataset API](#) using the following basic actions:

- `cache`
- `persist`
- `unpersist`

`cache` is simply `persist` with `MEMORY_AND_DISK` storage level.

```
// Cache Dataset -- it is lazy and so nothing really happens
val data = spark.range(1).cache

// Trigger caching by executing an action
// The common idiom is to execute count since it's fairly cheap
data.count
```

At this point you could use web UI's **Storage** tab to review the Datasets persisted. Visit <http://localhost:4040/storage>.



Storage

▼ RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
3	*(1) Range (0, 1, step=1, splits=8)	Memory Deserialized 1x Replicated	8	100%	352.0 B	0.0 B

Figure 1. web UI's Storage tab

`persist` uses [CacheManager](#) for an in-memory cache of structured queries (and [InMemoryRelation](#) logical operators), and is used to [cache structured queries](#) (which simply registers the structured queries as [InMemoryRelation](#) leaf logical operators).

At [withCachedData](#) phase (of execution of a structured query), `QueryExecution` requests the `CacheManager` to [replace segments of a logical query plan with their cached data](#) (including [subqueries](#)).

```
scala> println(data.queryExecution.withCachedData.numberedTreeString)
00 InMemoryRelation [id#9L], StorageLevel(disk, memory, deserialized, 1 replicas)
01     +- *(1) Range (0, 1, step=1, splits=8)
```

```

// Use the cached Dataset in another query
// Notice InMemoryRelation in use for cached queries
scala> df.withColumn("newId", 'id).explain(extended = true)
== Parsed Logical Plan ==
'Project [* , 'id AS newId#16]
+- Range (0, 1, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint, newId: bigint
Project [id#0L, id#0L AS newId#16L]
+- Range (0, 1, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#0L, id#0L AS newId#16L]
+- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
    +- *Range (0, 1, step=1, splits=Some(8))

== Physical Plan ==
*Project [id#0L, id#0L AS newId#16L]
+- InMemoryTableScan [id#0L]
    +- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
        +- *Range (0, 1, step=1, splits=Some(8))

// Clear in-memory cache using SQL
// Equivalent to spark.catalog.clearCache
scala> sql("CLEAR CACHE").collect
res1: Array[org.apache.spark.sql.Row] = Array()

// Visit http://localhost:4040/storage to confirm the cleaning

```

You can also use SQL's `CACHE TABLE [tableName]` to cache `tableName` table in memory. Unlike `cache` and `persist` operators, `CACHE TABLE` is an eager operation which is executed as soon as the statement is executed.

```
sql("CACHE TABLE [tableName]")
```

You could however use `LAZY` keyword to make caching lazy.

Note

```
sql("CACHE LAZY TABLE [tableName]")
```

Use SQL's `REFRESH TABLE [tableName]` to refresh a cached table.

Use SQL's `UNCACHE TABLE (IF EXISTS)? [tableName]` to remove a table from cache.

Use SQL's `CLEAR CACHE` to remove all tables from cache.

Be careful what you cache, i.e. what Dataset is cached, as it gives different queries cached.

Note

```
// cache after range(5)
val q1 = spark.range(5).cache.filter($"id" % 2 === 0).select("id")
scala> q1.explain
== Physical Plan ==
*Filter ((id#0L % 2) = 0)
+- InMemoryTableScan [id#0L], [(id#0L % 2) = 0]
    +- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
        +- *Range (0, 5, step=1, splits=8)

// cache at the end
val q2 = spark.range(1).filter($"id" % 2 === 0).select("id").cache
scala> q2.explain
== Physical Plan ==
InMemoryTableScan [id#17L]
    +- InMemoryRelation [id#17L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
        +- *Filter ((id#17L % 2) = 0)
            +- *Range (0, 1, step=1, splits=8)
```

You can check whether a Dataset was cached or not using the following code:

Tip

```
scala> :type q2
org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]

val cache = spark.sharedState.cacheManager
scala> cache.lookupCachedData(q2.queryExecution.logical).isDefined
res0: Boolean = false
```

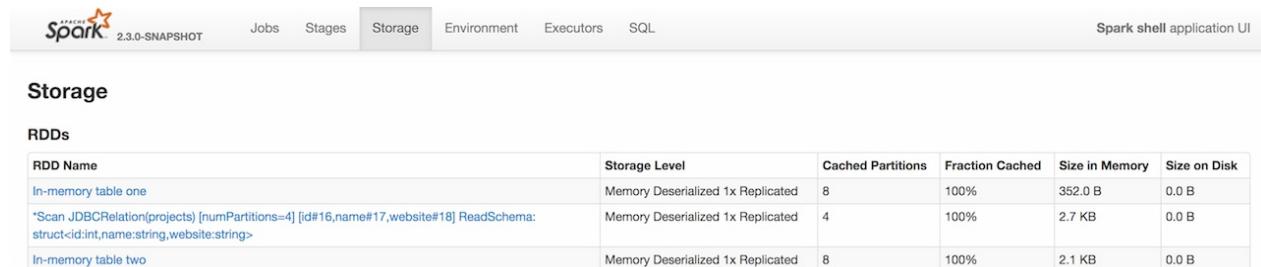
SQL's CACHE TABLE

SQL's `CACHE TABLE` corresponds to requesting the session-specific `catalog` to [caching the table](#).

Internally, `CACHE TABLE` becomes [CacheTableCommand](#) runnable command that...FIXME

User-Friendly Names Of Cached Queries in web UI's Storage Tab

As you may have noticed, web UI's Storage tab displays some [cached queries](#) with user-friendly RDD names (e.g. "In-memory table [name]") while others not (e.g. "Scan JDBCRelation...").



The screenshot shows the Spark Web UI interface. At the top, there is a navigation bar with tabs: Jobs, Stages, Storage (which is highlighted in grey), Environment, Executors, and SQL. To the right of the tabs, it says "Spark shell application UI". Below the navigation bar, the page title is "Storage". Under "RDDs", there is a table with the following data:

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
In-memory table one	Memory Deserialized 1x Replicated	8	100%	352.0 B	0.0 B
"Scan JDBCRelation(projects) [numPartitions=4] [id#16,name#17,website#18] ReadSchema: struct<id:int,name:string,website:string>"	Memory Deserialized 1x Replicated	4	100%	2.7 KB	0.0 B
In-memory table two	Memory Deserialized 1x Replicated	8	100%	2.1 KB	0.0 B

Figure 1. Cached Queries in web UI (Storage Tab)

"In-memory table [name]" RDD names are the result of SQL's [CACHE TABLE](#) or when [catalog](#) is requested to [cache a table](#).

```

// register Dataset as temporary view (table)
spark.range(1).createOrReplaceTempView("one")
// caching is lazy and won't happen until an action is executed
val one = spark.table("one").cache
// The following gives "*Range (0, 1, step=1, splits=8)"
// WHY?!
one.show

scala> spark.catalog.isCached("one")
res0: Boolean = true

one.unpersist

import org.apache.spark.storage.StorageLevel
// caching is lazy
spark.catalog.cacheTable("one", StorageLevel.MEMORY_ONLY)
// The following gives "In-memory table one"
one.show

spark.range(100).createOrReplaceTempView("hundred")
// SQL's CACHE TABLE is eager
// The following gives "In-memory table `hundred`"
// WHY single quotes?
spark.sql("CACHE TABLE hundred")

// register Dataset under name
val ds = spark.range(20)
spark.sharedState.cacheManager.cacheQuery(ds, Some("twenty"))
// trigger an action
ds.head

```

The other RDD names are due to [caching a Dataset](#).

```

val ten = spark.range(10).cache
ten.head

```

Dataset Checkpointing

Dataset Checkpointing is a feature of Spark SQL to truncate a logical query plan that could specifically be useful for highly iterative data algorithms (e.g. Spark MLlib that uses Spark SQL's `Dataset` API for data manipulation).

	<p>Checkpointing is actually a feature of Spark Core (that Spark SQL uses for distributed computations) that allows a driver to be restarted on failure with previously computed state of a distributed computation described as an <code>RDD</code>. That has been successfully used in Spark Streaming - the now-obsolete Spark module for stream processing based on RDD API.</p>
Note	<p>Checkpointing truncates the lineage of a RDD to be checkpointed. That has been successfully used in Spark MLlib in iterative machine learning algorithms like ALS.</p> <p>Dataset checkpointing in Spark SQL uses checkpointing to truncate the lineage of the underlying RDD of a <code>Dataset</code> being checkpointed.</p>

Checkpointing can be eager or lazy per `eager` flag of `checkpoint` operator. **Eager checkpointing** is the default checkpointing and happens immediately when requested. **Lazy checkpointing** does not and will only happen when an action is executed.

Using Dataset checkpointing requires that you specify the `checkpoint directory`. The directory stores the checkpoint files for RDDs to be checkpointed. Use `SparkContext.setCheckpointDir` to set the path to a checkpoint directory.

Checkpointing can be `local` or `reliable` which defines how reliable the `checkpoint directory` is. **Local checkpointing** uses executor storage to write checkpoint files to and due to the executor lifecycle is considered unreliable. **Reliable checkpointing** uses a reliable data storage like Hadoop HDFS.

Table 1. Dataset Checkpointing Types

	Eager	Lazy
Reliable	<code>checkpoint</code>	<code>checkpoint(eager = false)</code>
Local	<code>localCheckpoint</code>	<code>localCheckpoint(eager = false)</code>

A RDD can be recovered from a checkpoint files using `SparkContext.checkpointFile`. You can use `SparkSession.internalCreateDataFrame` method to (re)create the DataFrame from the RDD of internal binary rows.

Enable `INFO` logging level for `org.apache.spark.rdd.ReliableRDDCheckpointData` logger to see what happens while an RDD is checkpointed.

Add the following line to `conf/log4j.properties` :

Tip

```
log4j.logger.org.apache.spark.rdd.ReliableRDDCheckpointData=INFO
```

Refer to [Logging](#).

```
import org.apache.spark.sql.functions.rand
val nums = spark.range(5).withColumn("random", rand()).filter($"random" > 0.5)
scala> nums.show
+---+-----+
| id|      random|
+---+-----+
| 0| 0.752877642067488|
| 1|0.5271005540026181|
+---+-----+

scala> println(nums.queryExecution.toRdd.toDebugString)
(8) MapPartitionsRDD[7] at toRdd at <console>:27 []
 | MapPartitionsRDD[6] at toRdd at <console>:27 []
 | ParallelCollectionRDD[5] at toRdd at <console>:27 []

// Remember to set the checkpoint directory
scala> nums.checkpoint
org.apache.spark.SparkException: Checkpoint directory has not been set in the SparkContext
at org.apache.spark.rdd.RDD.checkpoint(RDD.scala:1548)
at org.apache.spark.sql.Dataset.checkpoint(Dataset.scala:594)
at org.apache.spark.sql.Dataset.checkpoint(Dataset.scala:539)
... 49 elided

spark.sparkContext.setCheckpointDir("/tmp/checkpoints")

val checkpointDir = spark.sparkContext.getCheckpointDir.get
scala> println(checkpointDir)
file:/tmp/checkpoints/b1f413dc-3eaf-46a0-99de-d795252035e0

val numsCheckpointed = nums.checkpoint
scala> println(numsCheckpointed.queryExecution.toRdd.toDebugString)
(8) MapPartitionsRDD[11] at toRdd at <console>:27 []
 | MapPartitionsRDD[9] at checkpoint at <console>:26 []
 | ReliableCheckpointRDD[10] at checkpoint at <console>:26 []

// Set org.apache.spark.rdd.ReliableRDDCheckpointData logger to INFO
// to see what happens while an RDD is checkpointed
// Let's use log4j API
import org.apache.log4j.{Level, Logger}
Logger.getLogger("org.apache.spark.rdd.ReliableRDDCheckpointData").setLevel(Level.INFO)
```

```

scala> nums.checkpoint
18/03/23 00:05:15 INFO ReliableRDDCheckpointData: Done checkpointing RDD 12 to file:/tmp/checkpoints/b1f413dc-3eaf-46a0-99de-d795252035e0/rdd-12, new parent is RDD 13
res7: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [id: bigint, random: double]

// Save the schema as it is going to use to reconstruct nums dataset from a RDD
val schema = nums.schema

// Recover nums dataset from the checkpoint files
// Start from recovering the underlying RDD
// And create a Dataset based on the RDD

// Get the path to the checkpoint files of the checkpointed RDD of the Dataset
import org.apache.spark.sql.execution.LogicalRDD
val logicalRDD = numsCheckpointed.queryExecution.optimizedPlan.asInstanceOf[LogicalRDD]
]
val checkpointFiles = logicalRDD.rdd.getCheckpointFile.get
scala> println(checkpointFiles)
file:/tmp/checkpoints/b1f413dc-3eaf-46a0-99de-d795252035e0/rdd-9

// SparkContext.checkpointFile is a `protected[spark]` method
// Use :paste -raw mode in Spark shell and define a helper object to "escape" the package lock-in
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package org.apache.spark
object my {
  import scala.reflect.ClassTag
  import org.apache.spark.rdd.RDD
  def recover[T: ClassTag](sc: SparkContext, path: String): RDD[T] = {
    sc.checkpointFile[T](path)
  }
}

// Exiting paste mode, now interpreting.

// Make sure to use the same checkpoint directory

import org.apache.spark.my
import org.apache.spark.sql.catalyst.InternalRow
val numsRddRecovered = my.recover[InternalRow](spark.sparkContext, checkpointFiles)
scala> :type numsRddRecovered
org.apache.spark.rdd.RDD[org.apache.spark.sql.catalyst.InternalRow]

// We have to convert RDD[InternalRow] to DataFrame

// Use :paste -raw again as we use `private[sql]` method
scala> :pa -raw
// Entering paste mode (ctrl-D to finish)

```

```

package org.apache.spark.sql
object my2 {
  import org.apache.spark.rdd.RDD
  import org.apache.spark.sql.{DataFrame, SparkSession}
  import org.apache.spark.sql.catalyst.InternalRow
  import org.apache.spark.sql.types.StructType
  def createDataFrame(spark: SparkSession, catalystRows: RDD[InternalRow], schema: StructType): DataFrame = {
    spark.internalCreateDataFrame(catalystRows, schema)
  }
}

// Exiting paste mode, now interpreting.

import org.apache.spark.sql.my2
val numsRecovered = my2.createDataFrame(spark, numsRddRecovered, schema)
scala> numsRecovered.show
+---+-----+
| id|      random|
+---+-----+
| 0| 0.752877642067488|
| 1|0.5271005540026181|
+---+-----+

```

Specifying Checkpoint Directory

— `SparkContext.setCheckpointDir` Method

```
SparkContext.setCheckpointDir(directory: String)
```

`setCheckpointDir` sets the [checkpoint directory](#).

Internally, `setCheckpointDir` ...FIXME

Recovering RDD From Checkpoint Files

— `SparkContext.checkpointFile` Method

```
SparkContext.checkpointFile(directory: String)
```

`checkpointFile` reads (*recovers*) a RDD from a checkpoint directory.

Note

`SparkContext.checkpointFile` is a `protected[spark]` method so the code to access it has to be in `org.apache.spark` package.

Internally, `checkpointFile` creates a `ReliableCheckpointRDD` in a scope.

UserDefinedAggregateFunction — Contract for User-Defined Untyped Aggregate Functions (UDAFs)

`UserDefinedAggregateFunction` is the contract to define **user-defined aggregate functions (UDAFs)**.

```
// Custom UDAF to count rows

import org.apache.spark.sql.Row
import org.apache.spark.sql.expressions.{MutableAggregationBuffer, UserDefinedAggregateFunction}
import org.apache.spark.sql.types.{DataType, LongType, StructType}

class MyCountUDAF extends UserDefinedAggregateFunction {
    override def inputSchema: StructType = {
        new StructType().add("id", LongType, nullable = true)
    }

    override def bufferSchema: StructType = {
        new StructType().add("count", LongType, nullable = true)
    }

    override def dataType: DataType = LongType

    override def deterministic: Boolean = true

    override def initialize(buffer: MutableAggregationBuffer): Unit = {
        println(s">>> initialize (buffer: $buffer)")
        // NOTE: Scala's update used under the covers
        buffer(0) = 0L
    }

    override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
        println(s">>> update (buffer: $buffer -> input: $input)")
        buffer(0) = buffer.getLong(0) + 1
    }

    override def merge(buffer: MutableAggregationBuffer, row: Row): Unit = {
        println(s">>> merge (buffer: $buffer -> row: $row)")
        buffer(0) = buffer.getLong(0) + row.getLong(0)
    }

    override def evaluate(buffer: Row): Any = {
        println(s">>> evaluate (buffer: $buffer)")
        buffer.getLong(0)
    }
}
```

UserDefinedAggregateFunction is created using [apply](#) or [distinct](#) factory methods.

```
val dataset = spark.range(start = 0, end = 4, step = 1, numPartitions = 2)

// Use the UDAF
val mycount = new MyCountUDAF
val q = dataset.
  withColumn("group", 'id % 2).
  groupBy('group).
  agg(mycount.distinct('id) as "count")
scala> q.show
+---+---+
|group|count|
+---+---+
|    0|    2|
|    1|    2|
+---+---+
```

The lifecycle of `UserDefinedAggregateFunction` is entirely managed using `ScalaUDAF` expression container.

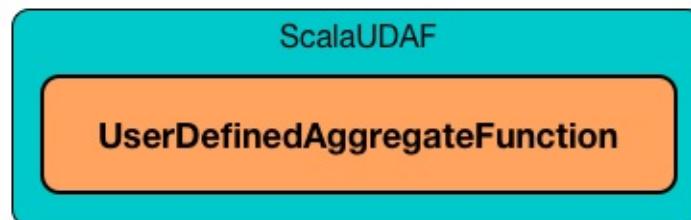


Figure 1. `UserDefinedAggregateFunction` and `ScalaUDAF` Expression Container

	<p>Use UDFRegistration to register a (temporary) <code>UserDefinedAggregateFunction</code> and use it in SQL mode.</p>
Note	<pre>import org.apache.spark.sql.expressions.UserDefinedAggregateFunction val mycount: UserDefinedAggregateFunction = ... spark.udf.register("mycount", mycount) spark.sql("SELECT mycount(*) FROM range(5)")</pre>

UserDefinedAggregateFunction Contract

```
package org.apache.spark.sql.expressions

abstract class UserDefinedAggregateFunction {
    // only required methods that have no implementation
    def bufferSchema: StructType
    def dataType: DataType
    def deterministic: Boolean
    def evaluate(buffer: Row): Any
    def initialize(buffer: MutableAggregationBuffer): Unit
    def inputSchema: StructType
    def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit
    def update(buffer: MutableAggregationBuffer, input: Row): Unit
}
```

Table 1. (Subset of) UserDefinedAggregateFunction Contract

Method	Description
bufferSchema	
dataType	
deterministic	
evaluate	
initialize	
inputSchema	
merge	
update	

Creating Column for UDAF — `apply` Method

```
apply(exprs: Column*): Column
```

`apply` creates a [Column](#) with [ScalaUDAF](#) (inside [AggregateExpression](#)).

Note	AggregateExpression uses complete mode and isDistinct flag is disabled.
------	---

```
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
val myUDAF: UserDefinedAggregateFunction = ...
val myUdafCol = myUDAF.apply($"id", $"name")
scala> myUdafCol.explain(extended = true)
mycountudaf('id, 'name, $line17.$read$$iw$$iw$MyCountUDAF@4704b66a, 0, 0)

scala> println(myUdafCol.expr.numberedTreeString)
00 mycountudaf('id, 'name, $line17.$read$$iw$$iw$MyCountUDAF@4704b66a, 0, 0)
01 +- MyCountUDAF('id,'name)
02   :- 'id
03   +- 'name

import org.apache.spark.sql.catalyst.expressions.aggregate.AggregateExpression
myUdafCol.expr.asInstanceOf[AggregateExpression]

import org.apache.spark.sql.execution.aggregate.ScalaUDAF
val scalaUdaf = myUdafCol.expr.children.head.asInstanceOf[ScalaUDAF]
scala> println(scalaUdaf.toString)
MyCountUDAF('id,'name)
```

Creating Column for UDAF with Distinct Values — `distinct` Method

```
distinct(exprs: Column*): Column
```

`distinct` creates a [Column](#) with [ScalaUDAF](#) (inside [AggregateExpression](#)).

Note `AggregateExpression` uses `Complete` mode and `isDistinct` flag is enabled.

Note `distinct` is like `apply` but has `isDistinct` flag enabled.

```
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
val myUDAF: UserDefinedAggregateFunction = ...
scala> val myUdafCol = myUDAF.distinct($"id", $"name")
myUdafCol: org.apache.spark.sql.Column = mycountudaf(DISTINCT id, name)

scala> myUdafCol.explain(extended = true)
mycountudaf(distinct 'id, 'name, $line17.$read$$iw$$iw$MyCountUDAF@4704b66a, 0, 0)

import org.apache.spark.sql.catalyst.expressions.aggregate.AggregateExpression
val aggExpr = myUdafCol.expr
scala> println(aggExpr.numberedTreeString)
00 mycountudaf(distinct 'id, 'name, $line17.$read$$iw$$iw$MyCountUDAF@4704b66a, 0, 0)
01 +- MyCountUDAF('id,'name)
02   :- 'id
03   +- 'name

scala> aggExpr.asInstanceOf[AggregateExpression].isDistinct
res0: Boolean = true
```

Aggregator—Contract for User-Defined Typed Aggregate Functions (UDAFs)

`Aggregator` is the [contract](#) for **user-defined typed aggregate functions** (aka *user-defined typed aggregations* or *UDAFs* in short).

```
package org.apache.spark.sql.expressions

abstract class Aggregator[-IN, BUF, OUT] extends Serializable {
    // only required methods that have no implementation
    def bufferEncoder: Encoder[BUF]
    def finish(reduction: BUF): OUT
    def merge(b1: BUF, b2: BUF): BUF
    def outputEncoder: Encoder[OUT]
    def reduce(b: BUF, a: IN): BUF
    def zero: BUF
}
```

After you create a custom `Aggregator`, you should use [toColumn](#) method to convert it to a `TypedColumn` that can be used with [Dataset.select](#) and [KeyValueGroupedDataset.agg](#) typed operators.

```
// From Spark MLlib's org.apache.spark.ml.recommendation.ALSModel
// Step 1. Create Aggregator
val topKAggregator: Aggregator[Int, Int, Float] = ???
val recs = ratings
.as[(Int, Int, Float)]
.groupByKey(_._1)
.agg(topKAggregator.toColumn) // <-- use the custom Aggregator
.toDF("id", "recommendations")
```

Note

Use `org.apache.spark.sql.expressions.scalalang.Typed` object to access the type-safe aggregate functions, i.e. `avg`, `count`, `sum` and `sumLong`.

```
import org.apache.spark.sql.expressions.scalalang.Typed
// Example 1
ds.groupByKey(_._1).agg(typed.sum(_._2))

// Example 2
ds.select(typed.sum((i: Int) => i))
```

Note	<p><code>Aggregator</code> is an <code>Experimental</code> and <code>Evolving</code> contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.</p> <p>In other words, using the contract is as treading on thin ice.</p>
------	---

`Aggregator` is used when:

- `SimpleTypedAggregateExpression` and `ComplexTypedAggregateExpression` are created
- `TypedAggregateExpression` is requested for the `aggregator`

Table 1. Aggregator Contract

Method	Description
<code>bufferEncoder</code>	Used when...FIXME
<code>finish</code>	Used when...FIXME
<code>merge</code>	Used when...FIXME
<code>outputEncoder</code>	Used when...FIXME
<code>reduce</code>	Used when...FIXME
<code>zero</code>	Used when...FIXME

Table 2. Aggregators

Aggregator	Description
<code>ParameterizedTypeSum</code>	
<code>ReduceAggregator</code>	
<code>TopByKeyAggregator</code>	Used exclusively in Spark MLlib
<code>TypedAverage</code>	
<code>TypedCount</code>	
<code>TypedSumDouble</code>	
<code>TypedSumLong</code>	

Converting Aggregator to TypedColumn — `toColumn` Method

```
toColumn: TypedColumn[IN, OUT]
```

`toColumn` ...FIXME

Note

`toColumn` is used when...FIXME

Configuration Properties

[Configuration properties](#) (aka *settings*) allow you to fine-tune a Spark SQL application.

You can set a configuration property in a [SparkSession](#) while creating a new instance using [config](#) method.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
  .master("local[*]")
  .appName("My Spark Application")
  .config("spark.sql.warehouse.dir", "c:/Temp") (1)
  .getOrCreate
```

1. Sets [spark.sql.warehouse.dir](#) for the Spark SQL session

You can also set a property using SQL `SET` command.

```
scala> spark.conf.getOption("spark.sql.hive.metastore.version")
res1: Option[String] = None

scala> spark.sql("SET spark.sql.hive.metastore.version=2.3.2").show(truncate = false)
+-----+-----+
|key          |value|
+-----+-----+
|spark.sql.hive.metastore.version|2.3.2|
+-----+-----+

scala> spark.conf.get("spark.sql.hive.metastore.version")
res2: String = 2.3.2
```

Table 1. Spark SQL Configuration Properties

Name	Description
<code>spark.sql.allowMultipleContexts</code>	Controls whether creating multiple SQLContexts/HiveContexts is allowed not (<code>false</code>) Default: <code>true</code>
	Maximum size (in bytes) for a table that broadcast to all worker nodes when performing a join. Default: <code>10L * 1024 * 1024</code> (10M)

<code>spark.sql.autoBroadcastJoinThreshold</code>	If the size of the statistics of the logical table is at most the setting, the DataFrame broadcast for join. Negative values or <code>0</code> disable broadcast joins. Use SQLConf.autoBroadcastJoinThreshold method to access the current value.
<code>spark.sql.avro.compression.codec</code>	The compression codec to use when writing data to disk Default: <code>snappy</code> The supported codecs are: <ul style="list-style-type: none">• <code>uncompressed</code>• <code>deflate</code>• <code>snappy</code>• <code>bzip2</code>• <code>xz</code> Use SQLConf.avroCompressionCodec method to access the current value.
<code>spark.sql.broadcastTimeout</code>	Timeout in seconds for the broadcast and broadcast joins. Default: <code>5 * 60</code> When negative, it is assumed infinite (<code>Duration.Inf</code>) Use SQLConf.broadcastTimeout method to access the current value.
<code>spark.sql.caseSensitive</code>	(internal) Controls whether the query should be case sensitive (<code>true</code>) or not. Default: <code>false</code> It is highly discouraged to turn on case mode. Use SQLConf.caseSensitiveAnalysis method to access the current value.
<code>spark.sql.cbo.enabled</code>	Enables cost-based optimization (CBC) estimation of plan statistics when <code>true</code> . Default: <code>false</code>

	Use SQLConf.cboEnabled method to access the current value.
<code>spark.sql.cbo.joinReorder.enabled</code>	<p>Enables join reorder for cost-based optimization (CBO).</p> <p>Default: <code>false</code></p> <p>Use SQLConf.joinReorderEnabled method to access the current value.</p>
<code>spark.sql.cbo.starSchemaDetection</code>	<p>Enables join reordering based on star schema detection for cost-based optimization (ReorderJoin logical plan optimization).</p> <p>Default: <code>false</code></p> <p>Use SQLConf.starSchemaDetection method to access the current value.</p>
<code>spark.sql.codegen.comments</code>	<p>Controls whether <code>CodegenContext</code> shows comments (<code>true</code>) or not (<code>false</code>).</p> <p>Default: <code>false</code></p>
<code>spark.sql.codegen.factoryMode</code>	<p>(internal) Determines the codegen fallback behavior</p> <p>Default: <code>FALLBACK</code></p> <p>Acceptable values:</p> <ul style="list-style-type: none"> • <code>CODEGEN_ONLY</code> - disable fallback mode • <code>FALLBACK</code> - try codegen first and, if compile error happens, fallback to interpreted mode • <code>NO_CODEGEN</code> - skips codegen and always takes the interpreted path <p>Used when <code>CodeGeneratorWithInterpreter</code> is requested to createObject (when <code>UnsafeProjection</code> is requested to createObject). See UnsafeProjection for Catalyst expressions.</p>
<code>spark.sql.codegen.fallback</code>	<p>(internal) Whether the whole stage can be temporary disabled for the part of a stage that has failed to compile generated code (<code>false</code>).</p> <p>Default: <code>true</code></p> <p>Use SQLConf.wholeStageFallback method to access the current value.</p>

	<p>(internal) The maximum bytecode size for a single compiled Java function generated by whole-stage codegen.</p> <p>Default: <code>65535</code></p> <p>The default value <code>65535</code> is the largest possible size for a valid Java method. On HotSpot, it may be preferable to set <code>8000</code> (which is the value of <code>HugeMethodLimit</code> in the OpenJDK JVM settings).</p> <p>Use SQLConf.hugeMethodLimit method to access the current value.</p>
<code>spark.sqlcodegen.useIdInClassName</code>	<p>(internal) Controls whether to embed the whole-stage codegen stage ID into the class name of the generated class as a suffix (<code>true</code>) or not (<code>false</code>).</p> <p>Default: <code>true</code></p> <p>Use SQLConf.wholeStageUseIdInClassName method to access the current value.</p>
<code>spark.sqlcodegen.maxFields</code>	<p>(internal) Maximum number of output fields (including nested fields) that whole-stage codegen supports. Going above the number defined here will result in error during whole-stage codegen.</p> <p>Default: <code>100</code></p> <p>Use SQLConf.wholeStageMaxNumFields method to access the current value.</p>
<code>spark.sqlcodegen.splitConsumeFuncByOperator</code>	<p>(internal) Controls whether whole stage codegen puts the logic of consuming rows of each operator into individual methods, instead of one big method. This can be used to avoid a function that can miss the opportunity for optimization.</p> <p>Default: <code>true</code></p> <p>Use SQLConf.wholeStageSplitConsumeFuncByOperator method to access the current value.</p>
<code>spark.sqlcodegen.wholeStage</code>	<p>(internal) Whether the whole stage (or its physical operators) will be compiled in a single Java method (<code>true</code>) or not (<code>false</code>).</p> <p>Default: <code>true</code></p> <p>Use SQLConf.wholeStageEnabled method to access the current value.</p>

	<p>(internal) Enables <code>OffHeapColumnVector</code> or <code>ColumnarBatch</code> (<code>true</code>) or not (<code>false</code>). <code>OnHeapColumnVector</code> is used if <code>false</code>.</p> <p>Default: <code>false</code></p> <p>Use <code>SQLConf.offHeapColumnVectorEnabled</code> method to access the current value.</p>
<code>spark.sql.columnNameOfCorruptRecord</code>	<p>(internal) Estimated size of a table or partition in query planning</p> <p>Default: Java's <code>Long.MaxValue</code></p> <p>Set to Java's <code>Long.MaxValue</code> which is <code>spark.sql.autoBroadcastJoinThreshold</code> times conservative. That is to say by default will not choose to broadcast a table unless for sure that the table size is small enough.</p> <p>Used by the planner to decide when it should broadcast a relation. By default, the system assumes that tables are too large to broadcast.</p> <p>Use <code>SQLConf.defaultSizeInBytes</code> method to access the current value.</p>
<code>spark.sql.dialect</code>	<p>(internal) When enabled (i.e. <code>true</code>), the planner will find duplicated exchanges in subqueries and re-use them.</p> <p>Default: <code>true</code></p>
<code>spark.sql.exchange.reuse</code>	<p>Note</p> <p>When disabled (i.e. <code>false</code>) <code>ReuseSubquery</code> and <code>ReuseExchange</code> physical optimizations (that the planner uses for physical query optimization) do nothing.</p> <p>Use <code>SQLConf.exchangeReuseEnabled</code> method to access the current value.</p>
<code>spark.sql.execution.useObjectHashAggregateExec</code>	<p>Enables <code>ObjectHashAggregateExec</code> when <code>Aggregation</code> execution planning strategy is executed.</p> <p>Default: <code>true</code></p>

	Use SQLConf.useObjectHashAggregate to access the current value.
<code>spark.sql.files.ignoreCorruptFiles</code>	<p>Controls whether to ignore corrupt files (not <code>false</code>). If <code>true</code>, the Spark jobs will continue to run when encountering corrupted files. The contents that have been read will still be included in the results.</p> <p>Default: <code>false</code></p> <p>Use SQLConf.ignoreCorruptFiles method to access the current value.</p>
<code>spark.sql.files.ignoreMissingFiles</code>	<p>Controls whether to ignore missing files (not <code>false</code>). If <code>true</code>, the Spark jobs will continue to run when encountering missing files. The contents that have been read will still be included in the results.</p> <p>Default: <code>false</code></p> <p>Use SQLConf.ignoreMissingFiles method to access the current value.</p>
<code>spark.sql.files.maxPartitionBytes</code>	<p>The maximum number of bytes to pack into a single partition when reading files.</p> <p>Default: <code>128 * 1024 * 1024</code> (which corresponds to <code>parquet.block.size</code>)</p> <p>Use SQLConf.filesMaxPartitionBytes method to access the current value.</p>
<code>spark.sql.files.openCostInBytes</code>	<p>(internal) The estimated cost to open a file measured by the number of bytes contained in it at the same time (to include multiple files in a single partition).</p> <p>Default: <code>4 * 1024 * 1024</code></p> <p>It's better to over estimate it, then the job will be faster than partitions with small files will be faster than partitions with large files (which is scheduled first).</p> <p>Use SQLConf.filesOpenCostInBytes method to access the current value.</p>
<code>spark.sql.join.preferSortMergeJoin</code>	<p>(internal) Controls whether JoinSelection planning strategy prefers sort merge join or shuffled hash join.</p> <p>Default: <code>true</code></p> <p>Use SQLConf.preferSortMergeJoin method to access the current value.</p>

<code>spark.sql.limit.scaleUpFactor</code>	<p>(internal) Minimal increase rate in the partitions between attempts when exec operator on a structured query. Higher values lead to more partitions read. Lower values lead to longer execution times as more jobs will be triggered.</p> <p>Default: 4</p> <p>Use SQLConf.limitScaleUpFactor method to access the current value.</p>
<code>spark.sql.optimizer.excludedRules</code>	<p>Comma-separated list of optimization rules that should be disabled (excluded) in the optimizer. The optimizer will log the rules that have been excluded.</p> <p>Default: (empty)</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>Note: It is not guaranteed that all rules listed in this configuration will eventually be excluded, as some rules are required for correctness.</p> </div> <p>Use SQLConf.optimizerExcludedRules method to access the current value.</p>
<code>spark.sql.optimizer.inSetConversionThreshold</code>	<p>(internal) The threshold of set size for conversion.</p> <p>Default: 10</p> <p>Use SQLConf.optimizerInSetConversionThreshold method to access the current value.</p>
<code>spark.sql.optimizer.maxIterations</code>	<p>Maximum number of iterations for Analyzer Optimizer.</p> <p>Default: 100</p>
<code>spark.sql.orc.impl</code>	<p>(internal) When native , use the native ORC support instead of the ORC library 1.2.1.</p> <p>Default: native</p> <p>Acceptable values:</p> <ul style="list-style-type: none"> • hive • native
	<p>Some other Parquet-producing systems, in particular Impala and older versions of HDFS, do not differentiate between binary dat</p>

<code>spark.sql.parquet.binaryAsString</code>	<p>when writing out the Parquet schema. Spark SQL to interpret binary data as ; provide compatibility with these system</p> <p>Default: <code>false</code></p> <p>Use SQLConf.isParquetBinaryAsString method to access the current value.</p>
<code>spark.sql.parquet.columnarReaderBatchSize</code>	<p>The number of rows to include in a par vectorized reader batch (the capacity o VectorizedParquetRecordReader).</p> <p>Default: <code>4096 (4k)</code></p> <p>The number should be carefully chose overhead and avoid OOMs while read</p> <p>Use SQLConf.parquetVectorizedRead method to access the current value.</p>
<code>spark.sql.parquet.int96AsTimestamp</code>	<p>Some Parquet-producing systems, in Impala, store Timestamp into INT96. S also store Timestamp as INT96 because avoid precision lost of the nanosecond flag tells Spark SQL to interpret INT96 timestamp to provide compatibility with systems.</p> <p>Default: <code>true</code></p> <p>Use SQLConf.isParquetINT96AsTime to access the current value.</p>
<code>spark.sql.parquet.enableVectorizedReader</code>	<p>Enables vectorized parquet decoding.</p> <p>Default: <code>true</code></p> <p>Use SQLConf.parquetVectorizedRead method to access the current value.</p>
<code>spark.sql.parquet.filterPushdown</code>	<p>Controls the filter predicate push-down for data sources using parquet file for</p> <p>Default: <code>true</code></p> <p>Use SQLConf.parquetFilterPushDown to access the current value.</p>
<code>spark.sql.parquet.filterPushdown.date</code>	<p>(internal) Enables parquet filter push-optimization for Date (when spark.sql.parquet.filterPushdown is en</p> <p>Default: <code>true</code></p>

	<p>Use SQLConf.parquetFilterPushDown to access the current value.</p>
<code>spark.sql.parquet.int96TimestampConversion</code>	<p>Controls whether timestamp adjustment is applied to INT96 data when converting timestamps, for data written by Impala.</p> <p>Default: <code>false</code></p> <p>This is necessary because Impala stores data with a different timezone offset than Spark.</p> <p>Use SQLConf.isParquetINT96TimestampConversion method to access the current value.</p>
	<p>Enables Parquet's native record-level filters.</p> <p>Default: <code>false</code></p>
<code>spark.sql.parquet.recordLevelFilter.enabled</code>	<p>Note This configuration only has effect when spark.sql.parquet.filterPushDown.enabled is enabled (and it is by default).</p> <p>Use SQLConf.parquetRecordFilterEnabled method to access the current value.</p>
<code>spark.sql.parser.quotedRegexColumnNames</code>	<p>Controls whether quoted identifiers (using <code>"</code>) in SELECT statements should be interpreted as regular expressions.</p> <p>Default: <code>false</code></p> <p>Use SQLConf.supportQuotedRegexColumnNames method to access the current value.</p>
	<p>(internal) Controls whether to use radix sort or not (<code>false</code>) in ShuffleExchangeExec and SortExec physical operators</p> <p>Default: <code>true</code></p> <p>Radix sort is much faster but requires more memory to be reserved up-front. The radix overhead may be significant when sorting rows (up to 50% more).</p> <p>Use SQLConf.enableRadixSort method to access the current value.</p>
	<p>(internal) Fully-qualified class name of FileCommitProtocol to use for...FIXME</p>

<code>spark.sql.sources.commitProtocolClass</code>	<p>Default: SQLHadoopMapReduceCom</p> <p>Use SQLConf.fileCommitProtocolClas access the current value.</p>
<code>spark.sql.sources.partitionOverwriteMode</code>	<p>Enables dynamic partition inserts when <code>INSERT OVERWRITE</code>.</p> <p>Default: <code>static</code></p> <p>When <code>INSERT OVERWRITE</code> a partitioned table with dynamic partition columns, Spark supports two modes (case-insensitive):</p> <ul style="list-style-type: none"> • static - Spark deletes all the partitions that match the partition specification (<code>PARTITION(a=1, b=2)</code>) in the <code>INSERT</code> before overwriting • dynamic - Spark doesn't delete partitions ahead, and only overwrites those that have data written into it <p>The default (STATIC) is to keep the same behavior of Spark prior to 2.3. Note that this correctly affects Hive serde tables, as they are always overwritten with dynamic mode.</p> <p>Use SQLConf.partitionOverwriteMode to access the current value.</p>
<code>spark.sql.pivotMaxValues</code>	<p>Maximum number of (distinct) values to collect without error (when doing a pivot) specifying the values for the pivot column.</p> <p>Default: <code>10000</code></p> <p>Use SQLConf.dataFramePivotMaxVal to access the current value.</p>
<code>spark.sql.redaction.options.regex</code>	<p>Regular expression to find options of a command with sensitive information.</p> <p>Default: <code>(?i)secret!password</code></p> <p>The values of the options matched will be redacted in the explain output.</p> <p>This redaction is applied on top of the redaction configuration defined by <code>spark.redaction.regex</code> configuration.</p> <p>Used exclusively when <code>SQLConf</code> is redacted via redactOptions.</p>
	<p>Regular expression to point at sensitive information in text output.</p>

	<p><code>spark.sql.redaction.string.regex</code></p> <p>Default: `(undefined)` When this regex string part, that string part is replaced by a dummy value (i.e. `****(redacted)`). This used to redact the output of SQL explain commands.</p>
	<p><code>Note</code> When this conf is not set, the <code>spark.redaction.string.regex</code> instead.</p>
	<p>Use SQLConf.stringRedactionPattern method to access the current value.</p>
<code>spark.sql.retainGroupColumns</code>	<p>Controls whether to retain columns used in aggregation or not (in RelationalGroupedDataset operators).</p> <p>Default: <code>true</code></p> <p>Use SQLConf.dataFrameRetainGroupedColumns method to access the current value.</p>
<code>spark.sql.runSQLOnFiles</code>	<p>(internal) Controls whether Spark SQL treats <code>datasource . path</code> as a table in a SQL query.</p> <p>Default: <code>true</code></p> <p>Use SQLConf.runSQLOnFile method to access the current value.</p>
<code>spark.sql.selfJoinAutoResolveAmbiguity</code>	<p>Controls whether to resolve ambiguity conditions for self-joins automatically (<code>false</code>)</p> <p>Default: <code>true</code></p>
<code>spark.sql.session.timeZone</code>	<p>The ID of session-local timezone, e.g. "America/Los_Angeles", etc.</p> <p>Default: Java's <code>TimeZone.getDefault().getID()</code></p> <p>Use SQLConf.sessionLocalTimeZone method to access the current value.</p>
<code>spark.sql.shuffle.partitions</code>	<p>Number of partitions to use by default for shuffling data for joins or aggregations</p> <p>Default: <code>200</code></p> <p>Corresponds to Apache Hive's <code>mapred.default.partition.size</code> property that Spark considers deprecated.</p> <p>Use SQLConf.numShufflePartitions method to access the current value.</p>

<code>spark.sql.sources.bucketing.enabled</code>	<p>Enables bucketing support. When disabled (<code>false</code>), bucketed tables are considered as (non-bucketed) tables.</p> <p>Default: <code>true</code></p> <p>Use SQLConf.bucketingEnabled method to get the current value.</p>
<code>spark.sql.sources.default</code>	<p>Defines the default data source to use DataFrameReader.</p> <p>Default: <code>parquet</code></p> <p>Used when:</p> <ul style="list-style-type: none"> • Reading (DataFrameWriter) or writing (DataFrameReader) datasets • Creating external table from a path (<code>Catalog.createExternalTable()</code>) • Reading (<code>DataStreamReader</code>) or writing (<code>DataStreamWriter</code>) in Structured
<code>spark.sql.subexpressionElimination.enabled</code>	<p>(internal) Enables subexpression elimination.</p> <p>Default: <code>true</code></p> <p>Use subexpressionEliminationEnabled method to access the current value.</p>
<code>spark.sql.ui.retainedExecutions</code>	<p>The number of SQLExecutionUIData entries to keep in <code>failedExecutions</code> and <code>completedExecutions</code> internal registries.</p> <p>Default: <code>1000</code></p> <p>When a query execution finishes, the entry is removed from the internal <code>activeExecutions</code> registry and stored in <code>failedExecutions</code> and <code>completedExecutions</code> given the end execution status. It is when <code>SQLListener</code> makes sure the number of <code>SQLExecutionUIData</code> entries exceed <code>spark.sql.ui.retainedExecutions</code> property and removes the excess of entries.</p>

Table 2. Spark SQL Configuration Properties (spark.sql.statistics)

Name	Description
	<p>Enables automatic calculation of statistics by falling back to HDFS if statistics are not available from table.</p> <p>Default: <code>false</code></p>

<code>s.s.s.fallBackToHdfs</code>	This can be useful in determining small enough for auto broadcast planning. Use SQLConf.fallBackToHdfsFor method to access the current value.		
<code>s.s.s.histogram.enabled</code>	Enables generating histograms with column statistics Default: <code>false</code> <table border="1"> <tr> <td>Note</td><td>Histograms can provide estimation accuracy. C Spark only supports exact histogram. Note that collecting histograms takes extra memory, for example, collecting column statistics usually takes a table scan, but generating height histogram will cause an extra table scan.</td></tr> </table> Use SQLConf.histogramEnabled method to access the current value.	Note	Histograms can provide estimation accuracy. C Spark only supports exact histogram. Note that collecting histograms takes extra memory, for example, collecting column statistics usually takes a table scan, but generating height histogram will cause an extra table scan.
Note	Histograms can provide estimation accuracy. C Spark only supports exact histogram. Note that collecting histograms takes extra memory, for example, collecting column statistics usually takes a table scan, but generating height histogram will cause an extra table scan.		
<code>s.s.s.histogram.numBins</code>	(internal) The number of bins when generating histograms. Default: <code>254</code> <table border="1"> <tr> <td>Note</td><td>The number of bins must be greater than 1.</td></tr> </table> Use SQLConf.histogramNumBins method to access the current value.	Note	The number of bins must be greater than 1.
Note	The number of bins must be greater than 1.		
<code>s.s.s.parallelFileListingInStatsComputation.enabled</code>	(internal) Enables parallel file listing commands, e.g. ANALYZE TABLE (to single thread listing that can be slow with tables with hundreds of millions of rows. Default: <code>true</code> Use SQLConf.parallelFileListingInStatsComputationEnabled method to access the current value.		
<code>spark.sql.statistics.ndv.maxError</code>	(internal) The maximum estimation error allowed in HyperLogLog++ algorithm for generating column level statistics Default: <code>0.05</code>		

	<p>spark.sql.statistics.percentile.accuracy</p>	<p>(internal) Accuracy of percentile when generating equi-height histograms. Larger value means better accuracy, relative error can be deduced by PERCENTILE_ACCURACY.</p> <p>Default: 10000</p>
	<p>s.s.s.size.autoUpdate.enabled</p>	<p>Enables automatic update of the statistic of a table after the table is modified.</p> <p>Default: false</p> <p>Important If the total number of rows in the table is very large, it may be expensive and time-consuming to update the data change continuously.</p> <p>Use SQLConf.autoSizeUpdateEnabled to access the current value.</p>

Table 3. Spark SQL Configuration Properties (spark.sql.adaptive)

Name	Description
s.s.adaptive.enabled	<p>Enables adaptive query execution</p> <p>Default: <code>false</code></p> <p>Use SQLConf.adaptiveExecutionEnabled method to access the current value.</p>
s.s.adaptive.minNumPostShufflePartitions	<p>(internal) The advisory minimal number of post-shuffle partitions for ExchangeCoordinator.</p> <p>Default: <code>-1</code></p> <p>This setting is used in Spark SQL tests to have enough parallelism to expose issues that will not be exposed with a single partition. Only positive values are used.</p> <p>Use SQLConf.minNumPostShufflePartitions method to access the current value.</p>
s.s.adaptive.shuffle.targetPostShuffleInputSize	<p>Recommended size of the input data of a post-shuffle partition (in bytes)</p> <p>Default: <code>64 * 1024 * 1024</code> bytes</p> <p>Use SQLConf.targetPostShuffleInputSize method to access the current value.</p>

Table 4. Spark SQL Configuration Properties (spark.sql.hive)

Name	Description
s.s.h.convertMetastoreOrc	<p>(internal) When enabled (i.e. <code>true</code>) the built-in ORC reader and writer are used to process ORC tables created using the HiveQL syntax (instead of Hive serde).</p> <p>Default: <code>true</code></p>
s.s.h.convertMetastoreParquet	<p>Controls whether to use the built-in Parquet reader and writer to process parquet tables created by using the HiveQL syntax (instead of Hive serde).</p> <p>Default: <code>true</code></p>

	<p><code>s.s.h.convertMetastoreParquet.mergeSchema</code></p> <p>Enables trying to merge possibly different but compatible Parquet schemas in different Parquet data files.</p> <p>Default: <code>false</code></p> <p>This configuration is only effective when <code>spark.sql.hive.convertMetastoreParquet</code> is enabled.</p>
	<p><code>s.s.h.manageFilesourcePartitions</code></p> <p>Enables metastore partition management for file source tables. This includes both datasource and converted Hive tables.</p> <p>Default: <code>true</code></p> <p>When enabled (<code>true</code>), datasource tables store partition in the Hive metastore, and use the metastore to prune partitions during query planning.</p> <p>Use <code>SQLConf.manageFilesourcePartitions</code> method to access the current value.</p>
	<p><code>s.s.h.metastore.barrierPrefixes</code></p> <p>Comma-separated list of class prefixes that should explicitly be reloaded for each version of Hive that Spark SQL is communicating with, e.g. Hive UDFs that are declared in a prefix that typically would be shared (i.e. <code>org.apache.spark.*</code>)</p> <p>Default: (empty)</p>
	<p><code>s.s.h.metastore.jars</code></p> <p>Location of the jars that should be used to create a HiveClientImpl.</p> <p>Default: <code>builtin</code></p> <p>Supported locations:</p> <ul style="list-style-type: none"> • <code>builtin</code> - the jars that were used to load Spark SQL (aka <i>Spark classes</i>). Valid only when using execution version of Hive, i.e. <code>spark.sql.hive.metastore.version</code> • <code>maven</code> - download the Hive jars from Maven repositories • Classpath in the standard format for both Hive and Hadoop

	<p>Comma-separated list of class prefixes that should be loaded using the classloader that is shared between Spark SQL and a specific version of Hive.</p> <p>Default: "com.mysql.jdbc", "org.postgresql", "com.microsoft.sqlserver", "oracle.jdbc"</p> <p><code>s.s.h.metastore.sharedPrefixes</code></p> <p>An example of classes that should be shared are:</p> <ul style="list-style-type: none"> • JDBC drivers that are needed to talk to the metastore • Other classes that interact with classes that are already shared e.g. custom appenders that are used by log4j
<code>s.s.h.metastore.version</code>	<p>Version of the Hive metastore (and the client classes and jars).</p> <p>Default: 1.2.1</p> <p>Supported versions range from 0.13.0 up to and including 2.3.2 .</p>
<code>s.s.h.verifyPartitionPath</code>	<p>When true, check all the partition paths under the table's root directory when reading data stored in HDFS. This configuration will be deprecated in the future releases and replaced by <code>spark.files.ignoreMissingFiles</code>.</p> <p>Default: false</p>
<code>s.s.h.metastorePartitionPruning</code>	
<code>s.s.h.filesourcePartitionFileCacheSize</code>	
<code>s.s.h.caseSensitiveInferenceMode</code>	
<code>s.s.h.convertCTAS</code>	
<code>s.s.h.gatherFastStats</code>	
<code>s.s.h.advancedPartitionPredicatePushdown.enabled</code>	

Table 5. Spark SQL Configuration Properties (spark.sql.inMemoryColumnarStorage)

Name	Description
s.s.i.batchSize	(internal) Controls...FIXME Default: 10000 Use SQLConf.columnBatchSize method to access the current value.
s.s.i.compressed	(internal) Controls...FIXME Default: true Use SQLConf.useCompression method to access the current value.
s.s.i.enableVectorizedReader	Enables vectorized reader for columnar caching. Default: true Use SQLConf.cacheVectorizedReaderEnabled method to access the current value.
s.s.i.partitionPruning	(internal) Enables partition pruning for in-memory columnar tables Default: true Use SQLConf.inMemoryPartitionPruning method to access the current value.

Table 6. Spark SQL Configuration Properties (spark.sql.windowExec)

Name	Description
s.s.w.buffer.in.memory.threshold	(internal) Threshold for number of rows guaranteed to be held in memory by WindowExec physical operator. Default: 4096 Use windowExecBufferInMemoryThreshold method to access the current value.
s.s.w.buffer.spill.threshold	(internal) Threshold for number of rows buffered in a WindowExec physical operator. Default: 4096 Use windowExecBufferSpillThreshold method to access the current value.

Catalog — Metastore Management Interface

`Catalog` is the [interface](#) for managing a **metastore** (aka *metadata catalog*) of relational entities (e.g. database(s), tables, functions, table columns and temporary views).

`Catalog` is available using [SparkSession.catalog](#) property.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.catalog
org.apache.spark.sql.catalog.Catalog
```

Table 1. Catalog Contract

Method	Description
<code>cacheTable</code>	<pre>cacheTable(tableName: String): Unit cacheTable(tableName: String, storageLevel: StorageLevel): Unit</pre> <p>↑ ↓</p> <p>Caches the specified table in memory Used for SQL's CACHE TABLE and AlterTableRenameCommand command.</p>
<code>clearCache</code>	<code>clearCache(): Unit</code>
<code>createTable</code>	<pre>createTable(tableName: String, path: String): DataFrame createTable(tableName: String, source: String, options: java.util.Map[String, String]): DataFrame createTable(tableName: String, source: String, options: Map[String, String]): DataFrame createTable(tableName: String, path: String, source: String): DataFrame createTable(tableName: String, source: String, schema: StructType, options: java.util.Map[String, String]): DataFrame createTable(tableName: String, source: String, schema: StructType, options: Map[String, String]): DataFrame</pre>

currentDatabase	currentDatabase: String
databaseExists	databaseExists(dbName: String): Boolean
dropGlobalTempView	dropGlobalTempView(viewName: String): Boolean
dropTempView	dropTempView(viewName: String): Boolean
functionExists	functionExists(functionName: String): Boolean functionExists(dbName: String, functionName: String): Boolean
getDatabase	getDatabase(dbName: String): Database
getFunction	getFunction(functionName: String): Function getFunction(dbName: String, functionName: String): Function
getTable	getTable(tableName: String): Table getTable(dbName: String, tableName: String): Table
isCached	isCached(tableName: String): Boolean
listColumns	listColumns(tableName: String): Dataset[Column] listColumns(dbName: String, tableName: String): Dataset[Column]
listDatabases	listDatabases(): Dataset[Database]
listFunctions	listFunctions(): Dataset[Function] listFunctions(dbName: String): Dataset[Function]

listTables	listTables(): Dataset[Table] listTables(dbName: String): Dataset[Table]
recoverPartitions	recoverPartitions(tableName: String): Unit
refreshByPath	refreshByPath(path: String): Unit
refreshTable	refreshTable(tableName: String): Unit
setCurrentDatabase	setCurrentDatabase(dbName: String): Unit
tableExists	tableExists(tableName: String): Boolean tableExists(dbName: String, tableName: String): Boolean
uncacheTable	uncacheTable(tableName: String): Unit

Note

[CatalogImpl](#) is the one and only known implementation of the [Catalog Contract](#) in Apache Spark.

CatalogImpl

`CatalogImpl` is the [Catalog](#) in Spark SQL that...FIXME

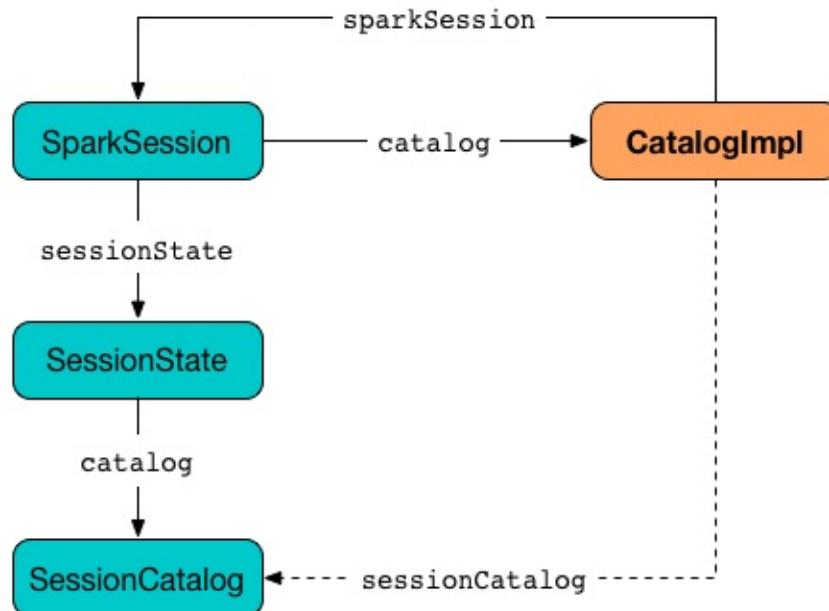


Figure 1. `CatalogImpl` uses `SessionCatalog` (through `SparkSession`)

Note

`CatalogImpl` is in `org.apache.spark.sql.internal` package.

Creating Table — `createTable` Method

```
createTable(
  tableName: String,
  source: String,
  schema: StructType,
  options: Map[String, String]): DataFrame
```

Note

`createTable` is part of [Catalog Contract](#) to...FIXME.

`createTable` ...FIXME

getTable Method

```
getTable(tableName: String): Table
getTable(dbName: String, tableName: String): Table
```

Note

`getTable` is part of [Catalog Contract](#) to...FIXME.

```
getTable ...FIXME
```

getFunction Method

```
getFunction(  
    functionName: String): Function  
getFunction(  
    dbName: String,  
    functionName: String): Function
```

Note

`getFunction` is part of Catalog Contract to...FIXME.

```
getFunction ...FIXME
```

functionExists Method

```
functionExists(  
    functionName: String): Boolean  
functionExists(  
    dbName: String,  
    functionName: String): Boolean
```

Note

`functionExists` is part of Catalog Contract to...FIXME.

```
functionExists ...FIXME
```

Caching Table or View In-Memory — cacheTable Method

```
cacheTable(tableName: String): Unit
```

Internally, `cacheTable` first creates a DataFrame for the table followed by requesting CacheManager to cache it.

Note

`cacheTable` uses the session-scoped SharedState to access the CacheManager .

Note

`cacheTable` is part of Catalog contract.

Removing All Cached Tables From In-Memory Cache — clearCache Method

```
clearCache(): Unit
```

`clearCache` requests `CacheManager` to remove all cached tables from in-memory cache.

Note

`clearCache` is part of [Catalog contract](#).

Creating External Table From Path — `createExternalTable` Method

```
createExternalTable(tableName: String, path: String): DataFrame
createExternalTable(tableName: String, path: String, source: String): DataFrame
createExternalTable(
  tableName: String,
  source: String,
  options: Map[String, String]): DataFrame
createExternalTable(
  tableName: String,
  source: String,
  schema: StructType,
  options: Map[String, String]): DataFrame
```

`createExternalTable` creates an external table `tableName` from the given `path` and returns the corresponding [DataFrame](#).

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

val readmeTable = spark.catalog.createExternalTable("readme", "README.md", "text")
readmeTable: org.apache.spark.sql.DataFrame = [value: string]

scala> spark.catalog.listTables.filter(_.name == "readme").show
+-----+-----+-----+-----+
| name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|readme| default|      null| EXTERNAL|     false|
+-----+-----+-----+-----+

scala> sql("select count(*) as count from readme").show(false)
+---+
|count|
+---+
| 99 |
+---+
```

The `source` input parameter is the name of the data source provider for the table, e.g. parquet, json, text. If not specified, `createExternalTable` uses `spark.sql.sources.default` setting to know the data source format.

Note	<code>source</code> input parameter must not be <code>hive</code> as it leads to a <code>AnalysisException</code> .
------	---

`createExternalTable` sets the mandatory `path` option when specified explicitly in the input parameter list.

`createExternalTable` parses `tableName` into `TableIdentifier` (using `SparkSqlParser`). It creates a `CatalogTable` and then `executes` (by `toRDD`) a `CreateTable` logical plan. The result `DataFrame` is a `Dataset[Row]` with the `QueryExecution` after executing `SubqueryAlias` logical plan and `RowEncoder`.

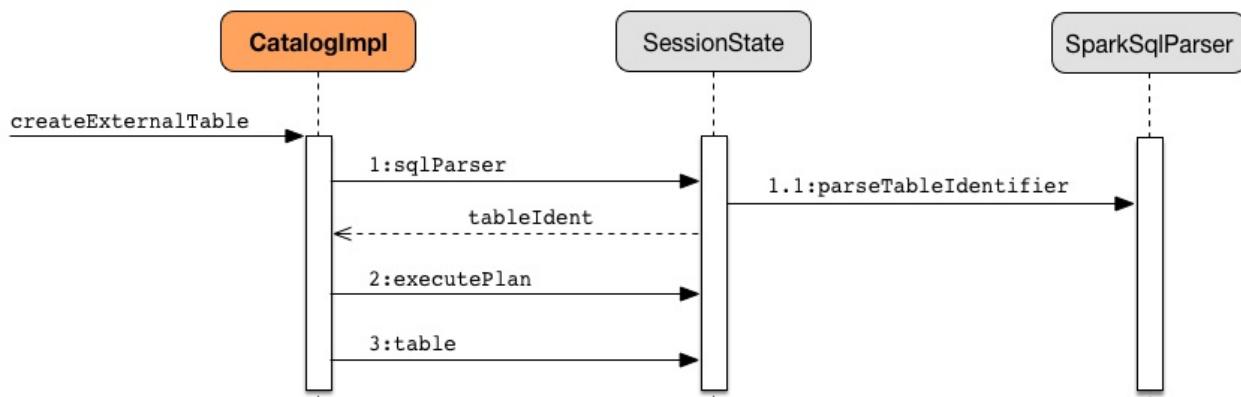


Figure 2. CatalogImpl.createExternalTable

Note	<code>createExternalTable</code> is part of Catalog contract .
------	--

Listing Tables in Database (as Dataset) — `listTables` Method

```

listTables(): Dataset[Table]
listTables(dbName: String): Dataset[Table]
  
```

Note	<code>listTables</code> is part of Catalog Contract to get a list of tables in the specified database.
------	--

Internally, `listTables` requests `SessionCatalog` to `list all tables` in the specified `dbName` database and `converts them to Tables`.

In the end, `listTables` `creates a Dataset` with the tables.

Listing Columns of Table (as Dataset) — `listColumns` Method

```
listColumns(tableName: String): Dataset[Column]
listColumns(dbName: String, tableName: String): Dataset[Column]
```

Note `listColumns` is part of [Catalog Contract](#) to...FIXME.

`listColumns` requests [SessionCatalog](#) for the `table` metadata.

`listColumns` takes the `schema` from the table metadata and creates a `Column` for every field (with the optional comment as the description).

In the end, `listColumns` creates a [Dataset](#) with the columns.

Converting TableIdentifier to Table — `makeTable` Internal Method

```
makeTable(tableIdent: TableIdentifier): Table
```

`makeTable` creates a `Table` using the input `TableIdentifier` and the `table` metadata (from the current [SessionCatalog](#)) if available.

Note `makeTable` uses [SparkSession](#) to access [SessionState](#) that is then used to access [SessionCatalog](#).

Note `makeTable` is used when `CatalogImpl` is requested to [listTables](#) or [getTable](#).

Creating Dataset from DefinedByConstructorParams Data — `makeDataset` Method

```
makeDataset[T <: DefinedByConstructorParams](
  data: Seq[T],
  sparkSession: SparkSession): Dataset[T]
```

`makeDataset` creates an [ExpressionEncoder](#) (from `DefinedByConstructorParams`) and encodes elements of the input `data` to internal binary rows.

`makeDataset` then creates a [LocalRelation](#) logical operator. `makeDataset` requests `SessionState` to execute the plan and creates the result `Dataset`.

Note `makeDataset` is used when `CatalogImpl` is requested to [list databases](#), [tables](#), [functions](#) and [columns](#)

Refreshing Analyzed Logical Plan of Table Query and Re-Caching It— `refreshTable` Method

```
refreshTable(tableName: String): Unit
```

Note

`refreshTable` is part of [Catalog Contract](#) to...FIXME.

`refreshTable` requests `SessionState` for the SQL parser to parse a `TableIdentifier` given the table name.

Note

`refreshTable` uses [SparkSession](#) to access the `SessionState`.

`refreshTable` requests [SessionCatalog](#) for the table metadata.

`refreshTable` then creates a `DataFrame` for the table name.

For a temporary or persistent `VIEW` table, `refreshTable` requests the analyzed logical plan of the `DataFrame` (for the table) to `refresh` itself.

For other types of table, `refreshTable` requests [SessionCatalog](#) for refreshing the table metadata (i.e. invalidating the table).

If the table has been cached, `refreshTable` requests `CacheManager` to `uncache` and `cache` the table `DataFrame` again.

Note

`refreshTable` uses [SparkSession](#) to access the `SharedState` that is used to access `CacheManager`.

refreshByPath Method

```
refreshByPath(resourcePath: String): Unit
```

Note

`refreshByPath` is part of [Catalog Contract](#) to...FIXME.

`refreshByPath` ...FIXME

listColumns Internal Method

```
listColumns(tableIdentifier: TableIdentifier): Dataset[Column]
```

`listColumns` ...FIXME

Note

`listColumns` is used exclusively when `CatalogImpl` is requested to [`listColumns`](#).

ExecutionListenerManager — Management Interface of QueryExecutionListeners

`ExecutionListenerManager` is the [management interface](#) for `QueryExecutionListeners` that listen for execution metrics:

- Name of the action (that triggered a query execution)
- [QueryExecution](#)
- Execution time of this query (in nanoseconds)

`ExecutionListenerManager` is available as [listenerManager](#) property of `SparkSession` (and [listenerManager](#) property of `SessionState`).

```
scala> :type spark.listenerManager
org.apache.spark.sql.util.ExecutionListenerManager

scala> :type spark.sessionState.listenerManager
org.apache.spark.sql.util.ExecutionListenerManager
```

`ExecutionListenerManager` takes a single `SparkConf` when created

While [created](#), `ExecutionListenerManager` reads `spark.sql.queryExecutionListeners` configuration property with `queryExecutionListeners` and [registers](#) them.

`ExecutionListenerManager` uses `spark.sql.queryExecutionListeners` configuration property as the list of `QueryExecutionListeners` that should be automatically added to newly created sessions (and registers them while [being created](#)).

Table 1. `ExecutionListenerManager`'s Public Methods

Method	Description
register	<code>register(listener: QueryExecutionListener): Unit</code>
unregister	<code>unregister(listener: QueryExecutionListener): Unit</code>
clear	<code>clear(): Unit</code>

`ExecutionListenerManager` is created exclusively when `BaseSessionStateBuilder` is requested for `ExecutionListenerManager` (while `SessionState` is built).

`ExecutionListenerManager` uses `listeners` internal registry for registered `QueryExecutionListeners`.

onSuccess Internal Method

```
onSuccess(funcName: String, qe: QueryExecution, duration: Long): Unit
```

onSuccess ...FIXME

	<p><code>onSuccess</code> is used when:</p> <ul style="list-style-type: none"> • <code>DataFrameWriter</code> is requested to run a logical command (after it has finished with no exceptions) • <code>Dataset</code> is requested to <code>withAction</code>
Note	

onFailure Internal Method

```
onFailure(funcName: String, qe: QueryExecution, exception: Exception): Unit
```

onFailure ...FIXME

	<p><code>onFailure</code> is used when:</p> <ul style="list-style-type: none"> • <code>DataFrameWriter</code> is requested to run a logical command (after it has reported an exception) • <code>Dataset</code> is requested to <code>withAction</code>
Note	

withErrorHandling Internal Method

```
withErrorHandling(f: QueryExecutionListener => Unit): Unit
```

withErrorHandling ...FIXME

	<p><code>withErrorHandling</code> is used when <code>ExecutionListenerManager</code> is requested to <code>onSuccess</code> and <code>onFailure</code>.</p>
Note	

Registering QueryExecutionListener — `register` Method

```
register(listener: QueryExecutionListener): Unit
```

Internally, `register` simply registers (adds) the input `QueryExecutionListener` to the `listeners` internal registry.

ExperimentalMethods

`ExperimentalMethods` holds extra [optimizations](#) and [strategies](#) that are used in [SparkOptimizer](#) and [SparkPlanner](#), respectively.

Table 1. ExperimentalMethods' Attributes

Name	Description
<code>extraOptimizations</code>	<p>Collection of rules to optimize LogicalPlans (i.e. <code>Rule[LogicalPlan]</code> objects)</p> <pre>extraOptimizations: Seq[Rule[LogicalPlan]]</pre>
<code>extraStrategies</code>	<p>Used when <code>SparkOptimizer</code> is requested for the User Provided Optimizers</p> <p>Collection of SparkStrategies</p> <pre>extraStrategies: Seq[Strategy]</pre> <p>Used when <code>SessionState</code> is requested for the SparkPlanner</p>

`ExperimentalMethods` is available as the [experimental](#) property of a `SparkSession`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.experimental
org.apache.spark.sql.ExperimentalMethods
```

Example

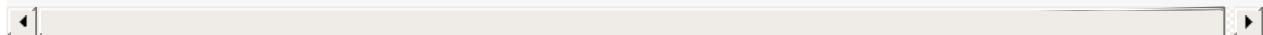
```
import org.apache.spark.sql.catalyst.rules.Rule
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan

object SampleRule extends Rule[LogicalPlan] {
    def apply(p: LogicalPlan): LogicalPlan = p
}

scala> :type spark
org.apache.spark.sql.SparkSession

spark.experimental.extraOptimizations = Seq(SampleRule)

// extraOptimizations is used in Spark Optimizer
val rule = spark.sessionState.optimizer.batches.flatMap(_.rules).filter(_ == SampleRule)
).head
scala> rule.ruleName
res0: String = SampleRule
```



ExternalCatalog Contract — External Catalog (Metastore) of Permanent Relational Entities

`ExternalCatalog` is the [contract](#) of an **external system catalog** (aka *metadata registry* or *metastore*) of permanent relational entities, i.e. databases, tables, partitions, and functions.

Table 1. ExternalCatalog's Features per Relational Entity

Feature	Database	Function	Partition	T
Alter	<code>alterDatabase</code>	<code>alterFunction</code>	<code>alterPartitions</code>	<code>alterTable</code> <code>alterTable</code> <code>alterTable</code>
Create	<code>createDatabase</code>	<code>createFunction</code>	<code>createPartitions</code>	<code>createTab</code>
Drop	<code>dropDatabase</code>	<code>dropFunction</code>	<code>dropPartitions</code>	<code>dropTable</code>
Get	<code>getDatabase</code>	<code>getFunction</code>	<code>getPartition</code> , <code>getPartitionOption</code>	<code>getTable</code>
List	<code>listDatabases</code>	<code>listFunctions</code>	<code>listPartitionNames</code> , <code>listPartitions</code> , <code>listPartitionsByFilter</code>	<code>listTables</code>
Load			<code>loadDynamicPartitions</code> , <code>loadPartition</code>	<code>loadTable</code>
Rename		<code>renameFunction</code>	<code>renamePartitions</code>	<code>renameTa</code>
Check Existence	<code>databaseExists</code>	<code>functionExists</code>		<code>tableExist</code>
Set				<code>setCurren</code>

Table 2. ExternalCatalog Contract (incl. Protected Methods)

Method	Description
<code>alterPartitions</code>	<code>alterPartitions(db: String, table: String, parts: Seq[CatalogTablePartition]): Unit</code>

createPartitions	<pre>createPartitions(db: String, table: String, parts: Seq[CatalogTablePartition], ignoreIfExists: Boolean): Unit</pre>
databaseExists	<pre>databaseExists(db: String): Boolean</pre>
doAlterDatabase	<pre>doAlterDatabase(dbDefinition: CatalogDatabase): Unit</pre>
doAlterFunction	<pre>doAlterFunction(db: String, funcDefinition: CatalogFunction)</pre>
doAlterTable	<pre>doAlterTable(tableDefinition: CatalogTable): Unit</pre>
doAlterTableDataSchema	<pre>doAlterTableDataSchema(db: String, table: String, newDataSchema: CatalogTableDataSchema): Unit</pre>
doAlterTableStats	<pre>doAlterTableStats(db: String, table: String, stats: Option[CatalogTableStats]): Unit</pre>
doCreateDatabase	<pre>doCreateDatabase(dbDefinition: CatalogDatabase, ignoreIfExists: Boolean): Unit</pre>
doCreateFunction	<pre>doCreateFunction(db: String, funcDefinition: CatalogFunction): Unit</pre>
doCreateTable	<pre>doCreateTable(tableDefinition: CatalogTable, ignoreIfExists: Boolean): Unit</pre>
doDropDatabase	<pre>doDropDatabase(db: String, ignoreIfNotExists: Boolean, cascade: Boolean): Unit</pre>
doDropFunction	<pre>doDropFunction(db: String, funcName: String): Unit</pre>

doDropTable	doDropTable(db: String, table: String, ignoreIfNotExists: Boolean, purge: Boolean): Unit
doRenameFunction	doRenameFunction(db: String, oldName: String, newName: String)
doRenameTable	doRenameTable(db: String, oldName: String, newName: String)
dropPartitions	dropPartitions(db: String, table: String, parts: Seq[TablePartitionSpec], ignoreIfNotExists: Boolean, purge: Boolean, retainData: Boolean): Unit
functionExists	functionExists(db: String, funcName: String): Boolean
getDatabase	getDatabase(db: String): CatalogDatabase
getFunction	getFunction(db: String, funcName: String): CatalogFunction
getPartition	getPartition(db: String, table: String, spec: TablePartitionSpec): CatalogTablePartition
getPartitionOption	getPartitionOption(db: String, table: String, spec: TablePartitionSpec): Option[CatalogTablePartition]
getTable	getTable(db: String, table: String): CatalogTable
listDatabases	listDatabases(): Seq[String] listDatabases(pattern: String): Seq[String]

<code>listFunctions</code>	<code>listFunctions(db: String, pattern: String): Seq[String]</code>
<code>listPartitionNames</code>	<code>listPartitionNames(db: String, table: String, partialSpec: Option[TablePartitionSpec] = None): Seq[String]</code>
<code>listPartitions</code>	<code>listPartitions(db: String, table: String, partialSpec: Option[TablePartitionSpec] = None): Seq[CatalogTablePartition]</code>
<code>listPartitionsByFilter</code>	<code>listPartitionsByFilter(db: String, table: String, predicates: Seq[Expression], defaultTimeKeyId: String): Seq[CatalogTablePartition]</code>
<code>listTables</code>	<code>listTables(db: String): Seq[String]</code> <code>listTables(db: String, pattern: String): Seq[String]</code>
<code>loadDynamicPartitions</code>	<code>loadDynamicPartitions(db: String, table: String, loadPath: String, partition: TablePartitionSpec, replace: Boolean, numDP: Int): Unit</code>
<code>loadPartition</code>	<code>loadPartition(db: String, table: String, loadPath: String, partition: TablePartitionSpec, isOverwrite: Boolean, inheritTableSpecs: Boolean, isSrcLocal: Boolean): Unit</code>
<code>loadTable</code>	<code>loadTable(db: String, table: String, loadPath: String, isOverwrite: Boolean, isSrcLocal: Boolean): Unit</code>

renamePartitions	renamePartitions(db: String, table: String, specs: Seq[TablePartitionSpec], newSpecs: Seq[TablePartitionSpec]): Unit
setCurrentDatabase	setCurrentDatabase(db: String): Unit
tableExists	tableExists(db: String, table: String): Boolean

`ExternalCatalog` is available as `externalCatalog` of `SharedState` (in `SparkSession`).

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sharedState.externalCatalog
org.apache.spark.sql.catalyst.catalog.ExternalCatalog
```

`ExternalCatalog` is available as ephemeral `in-memory` or persistent `hive-aware`.

Table 3. ExternalCatalogs

ExternalCatalog	Alias	Description
HiveExternalCatalog	hive	A persistent system catalog using a Hive metastore.
InMemoryCatalog	in-memory	An in-memory (ephemeral) system catalog that does not require setting up external systems (like a Hive metastore). It is intended for testing or exploration purposes only and therefore should not be used in production.

The concrete `ExternalCatalog` is chosen using `Builder.enableHiveSupport` that enables the Hive support (and sets `spark.sql.catalogImplementation` configuration property to `hive` when the Hive classes are available).

```

import org.apache.spark.sql.internal.StaticSQLConf
val catalogType = spark.conf.get(StaticSQLConf.CATALOG_IMPLEMENTATION.key)
scala> println(catalogType)
hive

scala> spark.sessionState.conf.getConf(StaticSQLConf.CATALOG_IMPLEMENTATION)
res1: String = hive

```

Tip Set `spark.sql.catalogImplementation` to `in-memory` when starting `spark-shell` to use [InMemoryCatalog](#) external catalog.

Tip

```

// spark-shell --conf spark.sql.catalogImplementation=in-memory
import org.apache.spark.sql.internal.StaticSQLConf
scala> spark.sessionState.conf.getConf(StaticSQLConf.CATALOG_IMPLEMENTATION)
res0: String = in-memory

```

Important You cannot change `ExternalCatalog` after `SparkSession` has been created using [spark.sql.catalogImplementation](#) configuration property as it is a static config.

Important

```

import org.apache.spark.sql.internal.StaticSQLConf
scala> spark.conf.set(StaticSQLConf.CATALOG_IMPLEMENTATION.key, "hive")
org.apache.spark.sql.AnalysisException: Cannot modify the value of a static
  park.sql.catalogImplementation;
    at org.apache.spark.sql.RuntimeConfig.requireNonStaticConf(RuntimeConfig.scala:41)
    at org.apache.spark.sql.RuntimeConfig.set(RuntimeConfig.scala:49)
    ... 49 elided

```

`ExternalCatalog` is a `ListenerBus` of `ExternalCatalogEventListener` listeners that handle `ExternalCatalogEvent` events.

Tip

Use `addListener` and `removeListener` to register and de-register `ExternalCatalogEventListener` listeners, accordingly.

Read [ListenerBus Event Bus Contract](#) in Mastering Apache Spark 2 gitbook to learn more about Spark Core's `ListenerBus` interface.

Altering Table Statistics — `alterTableStats` Method

```
alterTableStats(db: String, table: String, stats: Option[CatalogStatistics]): Unit
```

`alterTableStats` ...FIXME

Note

`alterTableStats` is used exclusively when `SessionCatalog` is requested for [altering the statistics of a table in a metastore](#) (that can happen when any logical command is executed that could change the table statistics).

Altering Table — `alterTable` Method

```
alterTable(tableDefinition: CatalogTable): Unit
```

`alterTable` ...FIXME

Note

`alterTable` is used exclusively when `SessionCatalog` is requested for [altering the statistics of a table in a metastore](#).

`createTable` Method

```
createTable(tableDefinition: CatalogTable, ignoreIfExists: Boolean): Unit
```

`createTable` ...FIXME

Note

`createTable` is used when...FIXME

`alterTableDataSchema` Method

```
alterTableDataSchema(db: String, table: String, newDataSchema: StructType): Unit
```

`alterTableDataSchema` ...FIXME

Note

`alterTableDataSchema` is used exclusively when `SessionCatalog` is requested to [alterTableDataSchema](#).

InMemoryCatalog

InMemoryCatalog is...FIXME

listPartitionsByFilter Method

```
listPartitionsByFilter(  
    db: String,  
    table: String,  
    predicates: Seq[Expression],  
    defaultTimeZoneId: String): Seq[CatalogTablePartition]
```

Note	listPartitionsByFilter is part of ExternalCatalog Contract to...FIXME.
------	--

listPartitionsByFilter ...FIXME

HiveExternalCatalog — Hive-Aware Metastore of Permanent Relational Entities

`HiveExternalCatalog` is a [external catalog of permanent relational entities](#) (aka *metastore*) that is used when `SparkSession` was created with [Hive support enabled](#).

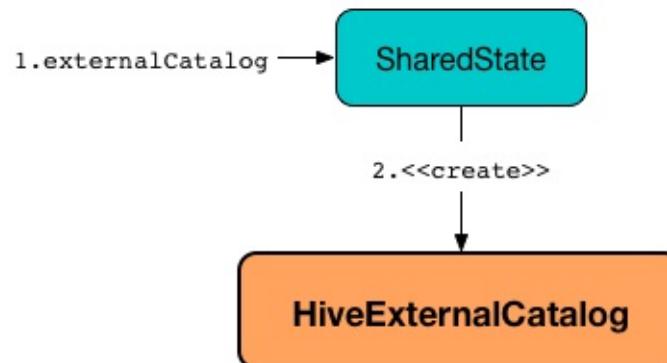


Figure 1. `HiveExternalCatalog` and `SharedState`

`HiveExternalCatalog` is [created](#) exclusively when `SharedState` is requested for the [ExternalCatalog](#) for the first time (and `spark.sql.catalogImplementation` internal configuration property is `hive`).

Note

The [Hadoop configuration](#) to create a `HiveExternalCatalog` is the default Hadoop configuration from Spark Core's `SparkContext.hadoopConfiguration` with the Spark properties with `spark.hadoop` prefix.

`HiveExternalCatalog` uses the internal [HiveClient](#) to retrieve metadata from a Hive metastore.

```

import org.apache.spark.sql.internal.StaticSQLConf
val catalogType = spark.conf.get(StaticSQLConf.CATALOG_IMPLEMENTATION.key)
scala> println(catalogType)
hive

// Alternatively...
scala> spark.sessionState.conf.getConf(StaticSQLConf.CATALOG_IMPLEMENTATION)
res1: String = hive

// Or you could use the property key by name
scala> spark.conf.get("spark.sql.catalogImplementation")
res1: String = hive

val metastore = spark.sharedState.externalCatalog
scala> :type metastore
org.apache.spark.sql.catalyst.catalog.ExternalCatalog

// Since Hive is enabled HiveExternalCatalog is the metastore
scala> println(metastore)
org.apache.spark.sql.hive.HiveExternalCatalog@25e95d04

```

Note

`spark.sql.catalogImplementation` configuration property is `in-memory` by default. Use `Builder.enableHiveSupport` to enable Hive support (that sets `spark.sql.catalogImplementation` internal configuration property to `hive` when the Hive classes are available).

```

import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
    .enableHiveSupport() // <-- enables Hive support
    .getOrCreate

```

Tip

Use `spark.sql.warehouse.dir` Spark property to change the location of Hive's `hive.metastore.warehouse.dir` property, i.e. the location of the Hive local/embedded metastore database (using Derby).

Refer to `SharedState` to learn about (the low-level details of) Spark SQL support for Apache Hive.

See also the official [Hive Metastore Administration](#) document.

Table 1. HiveExternalCatalog's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>client</code>	<code>HiveClient</code> for retrieving metadata from a Hive metastore Created by requesting <code>HiveUtils</code> for a new <code>HiveClientImpl</code> (with the current <code>SparkConf</code> and <code>Hadoop Configuration</code>)

getRawTable Method

```
getRawTable(db: String, table: String): CatalogTable
```

getRawTable ...FIXME

Note	getRawTable is used when...FIXME
------	----------------------------------

doAlterTableStats Method

```
doAlterTableStats(
  db: String,
  table: String,
  stats: Option[CatalogStatistics]): Unit
```

Note	doAlterTableStats is part of ExternalCatalog Contract to alter the statistics of a table.
------	---

doAlterTableStats ...FIXME

Converting Table Statistics to Properties — statsToProperties Internal Method

```
statsToProperties(
  stats: CatalogStatistics,
  schema: StructType): Map[String, String]
```

`statsToProperties` converts the [table statistics](#) to properties (i.e. key-value pairs that will be persisted as properties in the table metadata to a Hive metastore using the [Hive client](#)).

`statsToProperties` adds the following properties to the properties:

- **spark.sql.statistics.totalSize** with [total size \(in bytes\)](#)
- (if defined) **spark.sql.statistics.numRows** with [number of rows](#)

`statsToProperties` takes the [column statistics](#) and for every column (field) in `schema` converts the [column statistics to properties](#) and adds the properties (as [column statistic property](#)) to the properties.

Note

- `statsToProperties` is used when `HiveExternalCatalog` is requested for:
- `doAlterTableStats`
 - `alterPartitions`

Restoring Table Statistics from Properties (from Hive Metastore) — `statsFromProperties` Internal Method

```
statsFromProperties(
    properties: Map[String, String],
    table: String,
    schema: StructType): Option[CatalogStatistics]
```

`statsFromProperties` collects statistics-related `properties`, i.e. the properties with their keys with **spark.sql.statistics** prefix.

`statsFromProperties` returns `None` if there are no keys with the `spark.sql.statistics` prefix in `properties`.

If there are keys with `spark.sql.statistics` prefix, `statsFromProperties` creates a `ColumnStat` that is the column statistics for every column in `schema`.

For every column name in `schema` `statsFromProperties` collects all the keys that start with `spark.sql.statistics.colStats.[name]` prefix (after having checked that the key `spark.sql.statistics.colStats.[name].version` exists that is a marker that the column statistics exist in the statistics properties) and converts them to a `ColumnStat` (for the column name).

In the end, `statsFromProperties` creates a `CatalogStatistics` with the following properties:

- `sizeInBytes` as `spark.sql.statistics.totalSize` property
- `rowCount` as `spark.sql.statistics.numRows` property
- `colStats` as the collection of the column names and their `columnstat` (calculated above)

Note

`statsFromProperties` is used when `HiveExternalCatalog` is requested for restoring `table` and `partition` metadata.

`listPartitionsByFilter` Method

```
listPartitionsByFilter(
  db: String,
  table: String,
  predicates: Seq[Expression],
  defaultTimeKeyId: String): Seq[CatalogTablePartition]
```

Note `listPartitionsByFilter` is part of [ExternalCatalog Contract](#) to...FIXME.

`listPartitionsByFilter ...FIXME`

alterPartitions Method

```
alterPartitions(
  db: String,
  table: String,
  newParts: Seq[CatalogTablePartition]): Unit
```

Note `alterPartitions` is part of [ExternalCatalog Contract](#) to...FIXME.

`alterPartitions ...FIXME`

getTable Method

```
getTable(db: String, table: String): CatalogTable
```

Note `getTable` is part of [ExternalCatalog Contract](#) to...FIXME.

`getTable ...FIXME`

doAlterTable Method

```
doAlterTable(tableDefinition: CatalogTable): Unit
```

Note `doAlterTable` is part of [ExternalCatalog Contract](#) to alter a table.

`doAlterTable ...FIXME`

restorePartitionMetadata Internal Method

```
restorePartitionMetadata(
    partition: CatalogTablePartition,
    table: CatalogTable): CatalogTablePartition
```

`restorePartitionMetadata ...FIXME`

Note

- `getPartition`
- `getPartitionOption`

getPartition Method

```
getPartition(
    db: String,
    table: String,
    spec: TablePartitionSpec): CatalogTablePartition
```

Note

`getPartition` is part of [ExternalCatalog Contract](#) to...FIXME.

`getPartition ...FIXME`

getPartitionOption Method

```
getPartitionOption(
    db: String,
    table: String,
    spec: TablePartitionSpec): Option[CatalogTablePartition]
```

Note

`getPartitionOption` is part of [ExternalCatalog Contract](#) to...FIXME.

`getPartitionOption ...FIXME`

Creating HiveExternalCatalog Instance

`HiveExternalCatalog` takes the following when created:

- Spark configuration (i.e. `SparkConf`)
- Hadoop's [Configuration](#)

Building Property Name for Column and Statistic Key — `columnStatKeyPropName` Internal Method

```
columnStatKeyPropName(columnName: String, statKey: String): String
```

`columnStatKeyPropName` builds a property name of the form `spark.sql.statistics.colStats.[columnName].[statKey]` for the input `columnName` and `statKey`.

Note

`columnStatKeyPropName` is used when `HiveExternalCatalog` is requested to `statsToProperties` and `statsFromProperties`.

getBucketSpecFromTableProperties Internal Method

```
getBucketSpecFromTableProperties(metadata: CatalogTable): Option[BucketSpec]
```

`getBucketSpecFromTableProperties` ...FIXME

Note

`getBucketSpecFromTableProperties` is used when `HiveExternalCatalog` is requested to `restoreHiveSerdeTable` or `restoreDataSourceTable`.

Restoring Hive Serde Table — `restoreHiveSerdeTable` Internal Method

```
restoreHiveSerdeTable(table: CatalogTable): CatalogTable
```

`restoreHiveSerdeTable` ...FIXME

Note

`restoreHiveSerdeTable` is used exclusively when `HiveExternalCatalog` is requested to `restoreTableMetadata` (when there is no provider specified in table properties, which means this is a Hive serde table).

Restoring Data Source Table — `restoreDataSourceTable` Internal Method

```
restoreDataSourceTable(table: CatalogTable, provider: String): CatalogTable
```

`restoreDataSourceTable` ...FIXME

Note	<code>restoreDataSourceTable</code> is used exclusively when <code>HiveExternalCatalog</code> is requested to <code>restoreTableMetadata</code> (for regular data source table with provider specified in table properties).
------	--

restoreTableMetadata Internal Method

```
restoreTableMetadata(inputTable: CatalogTable): CatalogTable
```

`restoreTableMetadata` ...FIXME

Note	<code>restoreTableMetadata</code> is used when <code>HiveExternalCatalog</code> is requested for: <ul style="list-style-type: none"> • <code>getTable</code> • <code>doAlterTableStats</code> • <code>alterPartitions</code> • <code>listPartitionsByFilter</code>
------	--

Retrieving CatalogTablePartition of Table — listPartitions Method

```
listPartitions(  
  db: String,  
  table: String,  
  partialSpec: Option[TablePartitionSpec] = None): Seq[CatalogTablePartition]
```

Note	<code>listPartitions</code> is part of the ExternalCatalog Contract to list partitions of a table.
------	--

`listPartitions` ...FIXME

doCreateTable Method

```
doCreateTable(  
  tableDefinition: CatalogTable,  
  ignoreIfExists: Boolean): Unit
```

Note	<code>doCreateTable</code> is part of the ExternalCatalog Contract to...FIXME.
------	--

`doCreateTable` ...FIXME

tableMetaToTableProps Internal Method

```
tableMetaToTableProps(table: CatalogTable): mutable.Map[String, String]
tableMetaToTableProps(
  table: CatalogTable,
  schema: StructType): mutable.Map[String, String]
```

tableMetaToTableProps ...FIXME

Note	<code>tableMetaToTableProps</code> is used when <code>HiveExternalCatalog</code> is requested to <code>doAlterTableDataSchema</code> and <code>doCreateTable</code> (and <code>createDataSourceTable</code>).
------	--

doAlterTableDataSchema Method

```
doAlterTableDataSchema(
  db: String,
  table: String,
  newDataSchema: StructType): Unit
```

Note	<code>doAlterTableDataSchema</code> is part of the ExternalCatalog Contract to...FIXME.
------	---

doAlterTableDataSchema ...FIXME

createDataSourceTable Internal Method

```
createDataSourceTable(table: CatalogTable, ignoreIfExists: Boolean): Unit
```

createDataSourceTable ...FIXME

Note	<code>createDataSourceTable</code> is used exclusively when <code>HiveExternalCatalog</code> is requested to <code>doCreateTable</code> (for non-hive providers).
------	---

FunctionRegistry — Contract for Function Registries (Catalogs)

`FunctionRegistry` is the [contract](#) of function registries (*catalogs*) of native and user-defined functions.

```
package org.apache.spark.sql.catalyst.analysis

trait FunctionRegistry {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def clear(): Unit
    def dropFunction(name: FunctionIdentifier): Boolean
    def listFunction(): Seq[FunctionIdentifier]
    def lookupFunction(name: FunctionIdentifier): Option[ExpressionInfo]
    def lookupFunction(name: FunctionIdentifier, children: Seq[Expression]): Expression
    def lookupFunctionBuilder(name: FunctionIdentifier): Option[FunctionBuilder]
    def registerFunction(
        name: FunctionIdentifier,
        info: ExpressionInfo,
        builder: FunctionBuilder): Unit
}
```

Table 1. FunctionRegistry Contract

Property	Description
<code>clear</code>	Used exclusively when <code>SessionCatalog</code> is requested to reset
<code>dropFunction</code>	Used when...FIXME
<code>listFunction</code>	Used when...FIXME
<code>lookupFunction</code>	Used when: <ul style="list-style-type: none"> • <code>FunctionRegistry</code> is requested to functionExists • <code>SessionCatalog</code> is requested to find a function by name, lookupFunctionInfo or reset • <code>HiveSessionCatalog</code> is requested to lookupFunction0
<code>lookupFunctionBuilder</code>	Used when...FIXME
<code>registerFunction</code>	Used when: <ul style="list-style-type: none"> • <code>SessionCatalog</code> is requested to registerFunction or reset • <code>FunctionRegistry</code> is requested for a SimpleFunctionRegistry with the built-in functions registered or createOrReplaceTempFunction • <code>SimpleFunctionRegistry</code> is requested to <code>clone</code>

Note

The one and only `FunctionRegistry` available in Spark SQL is [SimpleFunctionRegistry](#).

`FunctionRegistry` is available through `functionRegistry` property of a `SessionState` (that is available as `sessionState` property of a `SparkSession`).

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.functionRegistry
org.apache.spark.sql.catalyst.analysis.FunctionRegistry
```

Note

You can register a new user-defined function using [UDFRegistration](#).

Table 2. FunctionRegistry's Attributes

Name	Description
builtin	SimpleFunctionRegistry with the built-in functions registered.

FunctionRegistry manages **function expression registry** of Catalyst expressions and the corresponding built-in/native SQL functions (that can be used in SQL statements).

Table 3. (Subset of) FunctionRegistry's Catalyst Expression to SQL Function Mapping

Catalyst Expression	SQL Function
CumeDist	cume_dist
IfNull	ifnull
Left	left
MonotonicallyIncreasingID	monotonically_increasing_id
NullIf	nullif
Nvl	nvl
Nvl2	nvl2
ParseToDate	to_date
ParseToTimestamp	to_timestamp
Right	right
CreateNamedStruct	struct

expression Internal Method

```
expression[T <: Expression](name: String)
  (implicit tag: ClassTag[T]): (String, (ExpressionInfo, FunctionBuilder))
```

expression ...FIXME

Note	expression is used when...FIXME
------	---------------------------------

SimpleFunctionRegistry

`SimpleFunctionRegistry` is the default [FunctionRegistry](#) that is backed by a hash map (with optional case sensitivity).

createOrReplaceTempFunction Final Method

```
createOrReplaceTempFunction(name: String, builder: FunctionBuilder): Unit
```

`createOrReplaceTempFunction` ...FIXME

Note	<code>createOrReplaceTempFunction</code> is used exclusively when <code>UDFRegistration</code> is requested to register an user-defined function , user-defined aggregate function , user-defined function (as UserDefinedFunction) or <code>registerPython</code> .
------	--

functionExists Method

```
functionExists(name: FunctionIdentifier): Boolean
```

`functionExists` ...FIXME

Note	<code>functionExists</code> is used when...FIXME
------	--

GlobalTempViewManager — Management Interface of Global Temporary Views

`GlobalTempViewManager` is the [interface](#) to manage global temporary views (that `SessionCatalog` uses when requested to [create](#), [alter](#) or [drop](#) global temporary views).

Strictly speaking, `GlobalTempViewManager` simply [manages](#) the names of the global temporary views registered (and the corresponding [logical plans](#)) and has no interaction with other services in Spark SQL.

`GlobalTempViewManager` is available as [globalTempViewManager](#) property of a `SharedState`.

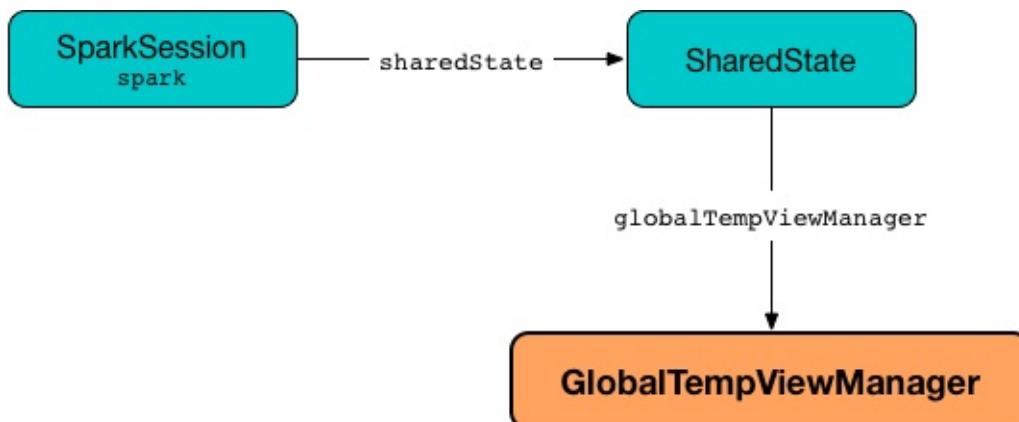


Figure 1. GlobalTempViewManager and SparkSession

```

scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sharedState.globalTempViewManager
org.apache.spark.sql.catalyst.catalog.GlobalTempViewManager
  
```

Table 1. GlobalTempViewManager API

Method	Description
clear	<code>clear(): Unit</code>
create	<code>create(name: String, viewDefinition: LogicalPlan, overrideIfExists: Boolean): Unit</code>
get	<code>get(name: String): Option[LogicalPlan]</code>
listViewNames	<code>listViewNames(pattern: String): Seq[String]</code>
remove	<code>remove(name: String): Boolean</code>
rename	<code>rename(oldName: String, newName: String): Boolean</code>
update	<code>update(name: String, viewDefinition: LogicalPlan): Boolean</code>

`GlobalTempViewManager` is `created` exclusively when `SharedState` is requested for `one` (for the very first time only as it is cached).

`GlobalTempViewManager` takes the name of the database when created.

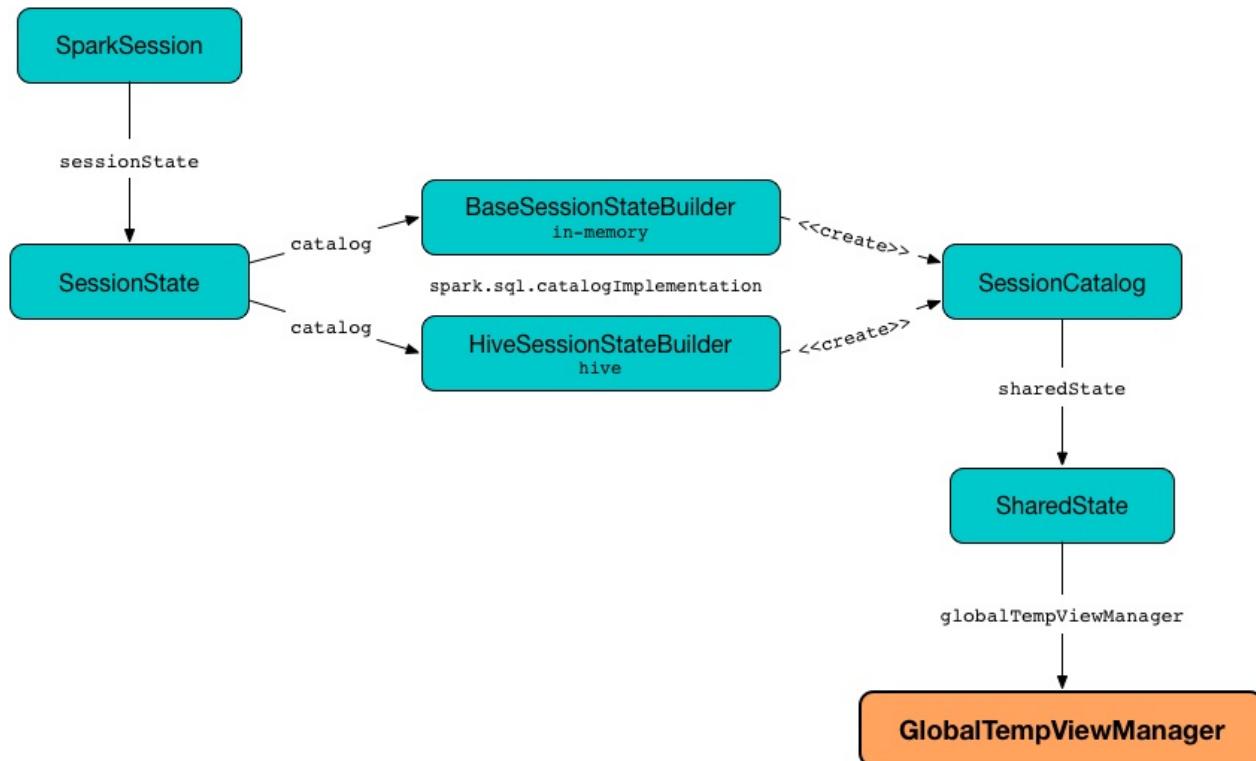


Figure 2. Creating GlobalTempViewManager

Table 2. GlobalTempViewManager's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
viewDefinitions	Registry of global temporary view definitions as logical plans per view name.

clear Method

```
clear(): Unit
```

`clear` simply removes all the entries in the `viewDefinitions` internal registry.

Note	<code>clear</code> is used when <code>SessionCatalog</code> is requested to <code>reset</code> (that happens to be exclusively in the Spark SQL internal tests).
------	--

Creating (Registering) Global Temporary View (Definition) — create Method

```
create(
  name: String,
  viewDefinition: LogicalPlan,
  overrideIfExists: Boolean): Unit
```

`create` simply registers (adds) the input [LogicalPlan](#) under the input `name`.

`create` throws an [AnalysisException](#) when the input `overrideIfExists` flag is off and the [viewDefinitions](#) internal registry contains the input `name`.

```
Temporary view '[table]' already exists
```

Note	<code>create</code> is used when <code>SessionCatalog</code> is requested to createGlobalTempView (when CreateViewCommand and CreateTempViewUsing logical commands are executed).
------	---

Retrieving Global View Definition Per Name — `get` Method

```
get(name: String): Option[LogicalPlan]
```

`get` simply returns the [LogicalPlan](#) that was registered under the `name` if it defined.

Note	<code>get</code> is used when <code>SessionCatalog</code> is requested to getGlobalTempView , getTempViewOrPermanentTableMetadata , lookupRelation , isTemporaryTable or refreshTable .
------	---

Listing Global Temporary Views For Pattern — `listViewNames` Method

```
listViewNames(pattern: String): Seq[String]
```

`listViewNames` simply gives a list of the global temporary views with names matching the input `pattern`.

Note	<code>listViewNames</code> is used exclusively when <code>SessionCatalog</code> is requested to listTables
------	--

Removing (De-Registering) Global Temporary View — `remove` Method

```
remove(name: String): Boolean
```

`remove` simply tries to remove the `name` from the [viewDefinitions](#) internal registry and returns `true` when removed or `false` otherwise.

Note

`remove` is used when `SessionCatalog` is requested to drop a [global temporary view or table](#).

rename Method

```
rename(oldName: String, newName: String): Boolean
```

`rename` ...FIXME

Note

`rename` is used when...FIXME

update Method

```
update(name: String, viewDefinition: LogicalPlan): Boolean
```

`update` ...FIXME

Note

`update` is used exclusively when `SessionCatalog` is requested to [alter a global temporary view](#).

SessionCatalog — Session-Spaced Catalog of Relational Entities

`SessionCatalog` is the catalog (*registry*) of relational entities, i.e. databases, tables, views, partitions, and functions (in a `SparkSession`).

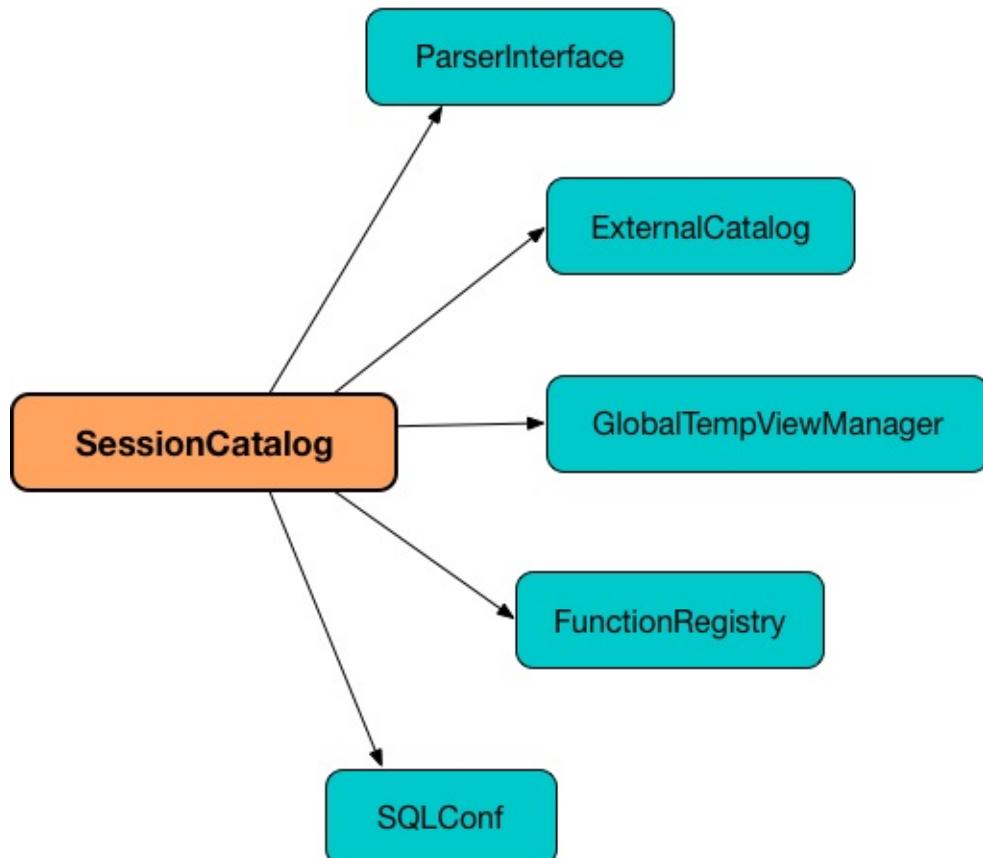


Figure 1. SessionCatalog and Spark SQL Services

`SessionCatalog` uses the `ExternalCatalog` for the metadata of permanent entities (i.e. `tables`).

Note

`SessionCatalog` is a layer over `ExternalCatalog` in a `SparkSession` which allows for different metastores (i.e. `in-memory` or `hive`) to be used.

`SessionCatalog` is available through `SessionState` (of a `SparkSession`).

```

scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.catalog
org.apache.spark.sql.catalyst.catalog.SessionCatalog
  
```

`SessionCatalog` is [created](#) when `BaseSessionStateBuilder` is requested for the `SessionCatalog` (when `SessionState` is requested for it).

Amongst the notable usages of `SessionCatalog` is to create an [Analyzer](#) or a [SparkOptimizer](#).

Table 1. SessionCatalog's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>currentDb</code>	FIXME Used when...FIXME
<code>tableRelationCache</code>	A cache of fully-qualified table names to table relation plans (i.e. <code>LogicalPlan</code>). Used when <code>SessionCatalog</code> refreshes a table
<code>tempViews</code>	Registry of temporary views (i.e. non-global temporary tables)

requireTableExists Internal Method

```
requireTableExists(name: TableIdentifier): Unit
```

`requireTableExists` ...FIXME

Note	<code>requireTableExists</code> is used when...FIXME
------	--

databaseExists Method

```
databaseExists(db: String): Boolean
```

`databaseExists` ...FIXME

Note	<code>databaseExists</code> is used when...FIXME
------	--

listTables Method

```
listTables(db: String): Seq[TableIdentifier] (1)
listTables(db: String, pattern: String): Seq[TableIdentifier]
```

1. Uses `"*"` as the pattern

```
listTables ...FIXME
```

Note	<p><code>listTables</code> is used when:</p> <ul style="list-style-type: none"> • <code>ShowTablesCommand</code> logical command is requested to run • <code>SessionCatalog</code> is requested to reset (for testing) • <code>CatalogImpl</code> is requested to listTables (for testing)
------	---

Checking Whether Table Is Temporary View

— `isTemporaryTable` Method

```
isTemporaryTable(name: TableIdentifier): Boolean
```

```
isTemporaryTable ...FIXME
```

Note	<code>isTemporaryTable</code> is used when...FIXME
------	--

`alterPartitions` Method

```
alterPartitions(tableName: TableIdentifier, parts: Seq[CatalogTablePartition]): Unit
```

```
alterPartitions ...FIXME
```

Note	<code>alterPartitions</code> is used when...FIXME
------	---

`listPartitions` Method

```
listPartitions(
  tableName: TableIdentifier,
  partialSpec: Option[TablePartitionSpec] = None): Seq[CatalogTablePartition]
```

```
listPartitions ...FIXME
```

Note	<code>listPartitions</code> is used when...FIXME
------	--

`alterTable` Method

```
alterTable(tableDefinition: CatalogTable): Unit
```

`alterTable ...FIXME`

Note

`alterTable` is used when `AlterTableSetPropertiesCommand`, `AlterTableUnsetPropertiesCommand`, `AlterTableChangeColumnCommand`, `AlterTableSetDePropertiesCommand`, `AlterTableRecoverPartitionsCommand`, `AlterTableSetLocationCommand`, `AlterViewAsCommand` (for permanent views) logical commands are executed.

Altering Table Statistics in Metastore (and Invalidating Internal Cache)— `alterTableStats` Method

```
alterTableStats(identifier: TableIdentifier, newStats: Option[CatalogStatistics]): Unit
```

`alterTableStats` requests `ExternalCatalog` to alter the statistics of the table (per `identifier`) followed by invalidating the table relation cache.

`alterTableStats` reports a `NoSuchDatabaseException` if the database does not exist.

`alterTableStats` reports a `NoSuchTableException` if the table does not exist.

Note

`alterTableStats` is used when the following logical commands are executed:

- `AnalyzeTableCommand`, `AnalyzeColumnCommand`, `AlterTableAddPartitionCommand`, `TruncateTableCommand`
- (indirectly through `commandUtils` when requested for updating existing table statistics) `InsertIntoHiveTable`, `InsertIntoHadoopFsRelationCommand`, `AlterTableDropPartitionCommand`, `AlterTableSetLocationCommand` and `LoadDataCommand`

tableExists Method

```
tableExists(name: TableIdentifier): Boolean
```

`tableExists ...FIXME`

Note

`tableExists` is used when...FIXME

functionExists Method

```
functionExists(name: FunctionIdentifier): Boolean
```

```
functionExists ...FIXME
```

Note

`functionExists` is used in:

- [LookupFunctions](#) logical rule (to make sure that [UnresolvedFunction](#) can be resolved, i.e. is registered with `SessionCatalog`)
- `CatalogImpl` to [check if a function exists in a database](#)
- ...

listFunctions Method

```
listFunctions(  
    db: String): Seq[(FunctionIdentifier, String)]  
listFunctions(  
    db: String,  
    pattern: String): Seq[(FunctionIdentifier, String)]
```

```
listFunctions ...FIXME
```

Note

`listFunctions` is used when...FIXME

InValidating Table Relation Cache (aka Refreshing Table) — refreshTable Method

```
refreshTable(name: TableIdentifier): Unit
```

```
refreshTable ...FIXME
```

Note

`refreshTable` is used when...FIXME

loadFunctionResources Method

```
loadFunctionResources(resources: Seq[FunctionResource]): Unit
```

```
loadFunctionResources ...FIXME
```

Note

`loadFunctionResources` is used when...FIXME

Altering (Updating) Temporary View (Logical Plan) — `alterTempViewDefinition` Method

```
alterTempViewDefinition(name: TableIdentifier, viewDefinition: LogicalPlan): Boolean
```

`alterTempViewDefinition` alters the temporary view by [updating an in-memory temporary table](#) (when a database is not specified and the table has already been registered) or a global temporary table (when a database is specified and it is for global temporary tables).

Note	"Temporary table" and "temporary view" are synonyms.
------	--

`alterTempViewDefinition` returns `true` when an update could be executed and finished successfully.

Note	<code>alterTempViewDefinition</code> is used exclusively when <code>AlterViewAsCommand</code> logical command is executed .
------	---

Creating (Registering) Or Replacing Local Temporary View — `createTempView` Method

```
createTempView(  
    name: String,  
    tableDefinition: LogicalPlan,  
    overrideIfExists: Boolean): Unit
```

`createTempView` ...FIXME

Note	<code>createTempView</code> is used when...FIXME
------	--

Creating (Registering) Or Replacing Global Temporary View — `createGlobalTempView` Method

```
createGlobalTempView(  
    name: String,  
    viewDefinition: LogicalPlan,  
    overrideIfExists: Boolean): Unit
```

`createGlobalTempView` simply requests the [GlobalTempViewManager](#) to [create a global temporary view](#).

Note	<p><code>createGlobalTempView</code> is used when:</p> <ul style="list-style-type: none"> • <code>CreateViewCommand</code> logical command is executed (for a global temporary view, i.e. when the view type is GlobalTempView) • <code>CreateTempViewUsing</code> logical command is executed (for a global temporary view, i.e. when the global flag is on)
------	---

createTable Method

```
createTable(tableDefinition: CatalogTable, ignoreIfExists: Boolean): Unit
```

`createTable` ...FIXME

Note	<code>createTable</code> is used when...FIXME
------	---

Creating SessionCatalog Instance

`SessionCatalog` takes the following when created:

- [ExternalCatalog](#)
- [GlobalTempViewManager](#)
- [FunctionResourceLoader](#)
- [FunctionRegistry](#)
- [CatalystConf](#)
- Hadoop's [Configuration](#)
- [ParserInterface](#)

`SessionCatalog` initializes the [internal registries and counters](#).

Finding Function by Name (Using FunctionRegistry) — lookupFunction Method

```
lookupFunction(  
    name: FunctionIdentifier,  
    children: Seq[Expression]): Expression
```

`lookupFunction` finds a function by `name`.

For a function with no database defined that exists in `FunctionRegistry`, `lookupFunction` requests `FunctionRegistry` to find the function (by its unqualified name, i.e. with no database).

If the `name` function has the database defined or does not exist in `FunctionRegistry`, `lookupFunction` uses the fully-qualified function `name` to check if the function exists in `FunctionRegistry` (by its fully-qualified name, i.e. with a database).

For other cases, `lookupFunction` requests `ExternalCatalog` to find the function and loads its resources. It then creates a corresponding temporary function and looks up the function again.

Note	<p><code>lookupFunction</code> is used when:</p> <ul style="list-style-type: none"> • <code>ResolveFunctions</code> logical resolution rule is executed (and resolves <code>UnresolvedGenerator</code> or <code>UnresolvedFunction</code> expressions) • <code>HiveSessionCatalog</code> is requested to <code>lookupFunction0</code>
------	---

Finding Relation (Table or View) in Catalogs

— `lookupRelation` Method

```
lookupRelation(name: TableIdentifier): LogicalPlan
```

`lookupRelation` finds the `name` table in the catalogs (i.e. `GlobalTempViewManager`, `ExternalCatalog` or registry of temporary views) and gives a `SubqueryAlias` per table type.

```
scala> :type spark.sessionState.catalog
org.apache.spark.sql.catalyst.catalog.SessionCatalog

import spark.sessionState.{catalog => c}
import org.apache.spark.sql.catalyst.TableIdentifier

// Global temp view
val db = spark.sharedState.globalTempViewManager.database
// Make the example reproducible (and so "replace")
spark.range(1).createOrReplaceGlobalTempView("gv1")
val gv1 = TableIdentifier(table = "gv1", database = Some(db))
val plan = c.lookupRelation(gv1)
scala> println(plan.numberedTreeString)
00 SubqueryAlias gv1
01 +- Range (0, 1, step=1, splits=Some(8))

val metastore = spark.sharedState.externalCatalog

// Regular table
val db = spark.catalog.currentDatabase
```

```

metastore.dropTable(db, table = "t1", ignoreIfNotExists = true, purge = true)
sql("CREATE TABLE t1 (id LONG) USING parquet")
val t1 = TableIdentifier(table = "t1", database = Some(db))
val plan = c.lookupRelation(t1)
scala> println(plan.numberedTreeString)
00 'SubqueryAlias t1
01 +- 'UnresolvedCatalogRelation `default`.'t1`, org.apache.hadoop.hive.ql.io.parquet.
serde.ParquetHiveSerDe

// Regular view (not temporary view!)
// Make the example reproducible
metastore.dropTable(db, table = "v1", ignoreIfNotExists = true, purge = true)
import org.apache.spark.sql.catalyst.catalog.{CatalogStorageFormat, CatalogTable, Cata
logTableType}
val v1 = TableIdentifier(table = "v1", database = Some(db))
import org.apache.spark.sql.types.StructType
val schema = new StructType().add($"id".long)
val storage = CatalogStorageFormat(locationUri = None, inputFormat = None, outputForma
t = None, serde = None, compressed = false, properties = Map())
val tableDef = CatalogTable(
  identifier = v1,
  tableType = CatalogTableType.VIEW,
  storage,
  schema,
  viewText = Some("SELECT 1") /* Required or RuntimeException reported */
metastore.createTable(tableDef, ignoreIfExists = false)
val plan = c.lookupRelation(v1)
scala> println(plan.numberedTreeString)
00 'SubqueryAlias v1
01 +- View (`default`.'v1`, [id#77L])
02   +- 'Project [unresolvedalias(1, None)]
03     +- OneRowRelation

// Temporary view
spark.range(1).createOrReplaceTempView("v2")
val v2 = TableIdentifier(table = "v2", database = None)
val plan = c.lookupRelation(v2)
scala> println(plan.numberedTreeString)
00 SubqueryAlias v2
01 +- Range (0, 1, step=1, splits=Some(8))

```

Internally, `lookupRelation` looks up the `name` table using:

1. [GlobalTempViewManager](#) when the database name of the table matches the `name` of `GlobalTempViewManager`
 - i. Gives `SubqueryAlias` or reports a `NoSuchTableException`
2. [ExternalCatalog](#) when the database name of the table is specified explicitly or the registry of temporary views does not contain the table
 - i. Gives `SubqueryAlias` with `view` when the table is a view (aka *temporary table*)

ii. Gives `SubqueryAlias` with `UnresolvedCatalogRelation` otherwise

3. The [registry of temporary views](#)

i. Gives `SubqueryAlias` with the logical plan per the table as registered in the [registry of temporary views](#)

Note

`lookupRelation` considers **default** to be the name of the database if the `name` table does not specify the database explicitly.

Note

`lookupRelation` is used when:

- `DescribeTableCommand` logical command is [executed](#)
- `ResolveRelations` logical evaluation rule is requested to [lookupTableFromCatalog](#)

Retrieving Table Metadata from External Catalog (Metastore) — `getTableMetadata` Method

```
getTableMetadata(name: TableIdentifier): CatalogTable
```

`getTableMetadata` simply requests [external catalog](#) (metastore) for the [table metadata](#).

Before requesting the external metastore, `getTableMetadata` makes sure that the [database](#) and [table](#) (of the input `TableIdentifier`) both exist. If either does not exist,

`getTableMetadata` reports a `NoSuchDatabaseException` or `NoSuchTableException`, respectively.

Retrieving Table Metadata — `getTempViewOrPermanentTableMetadata` Method

```
getTempViewOrPermanentTableMetadata(name: TableIdentifier): CatalogTable
```

Internally, `getTempViewOrPermanentTableMetadata` branches off per database.

When a database name is not specified, `getTempViewOrPermanentTableMetadata` finds a local [temporary view](#) and creates a [CatalogTable](#) (with `VIEW` [table type](#) and an undefined [storage](#)) or retrieves the table metadata from an [external catalog](#).

With the database name of the [GlobalTempViewManager](#),

`getTempViewOrPermanentTableMetadata` requests [GlobalTempViewManager](#) for the [global view definition](#) and creates a [CatalogTable](#) (with the `name` of `GlobalTempViewManager` in `table`

`identifier`, `VIEW` `table type` and an undefined `storage`) or reports a `NoSuchTableException`.

With the database name not of `GlobalTempViewManager`,

`getTempViewOrPermanentTableMetadata` simply retrieves the table metadata from an external catalog.

Note

`getTempViewOrPermanentTableMetadata` is used when:

- `CatalogImpl` is requested for converting `TableIdentifier` to `Table`, listing the columns of a table (as `Dataset`) and refreshing a table (i.e. the analyzed logical plan of the table query and re-caching it)
- `AlterTableAddColumnsCommand`, `CreateTableLikeCommand`, `DescribeColumnCommand`, `ShowColumnsCommand` and `ShowTablesCommand` logical commands are requested to run (executed)

Reporting `NoSuchDatabaseException` When Specified Database Does Not Exist — `requireDbExists` Internal Method

```
requireDbExists(db: String): Unit
```

`requireDbExists` reports a `NoSuchDatabaseException` if the specified database does not exist. Otherwise, `requireDbExists` does nothing.

reset Method

```
reset(): Unit
```

`reset` ...FIXME

Note

`reset` is used exclusively in the Spark SQL internal tests.

Dropping Global Temporary View — `dropGlobalTempView` Method

```
dropGlobalTempView(name: String): Boolean
```

`dropGlobalTempView` simply requests the `GlobalTempViewManager` to remove the `name` global temporary view.

Note	<code>dropGlobalTempView</code> is used when...FIXME
------	--

Dropping Table — `dropTable` Method

```
dropTable(  
    name: TableIdentifier,  
    ignoreIfNotExists: Boolean,  
    purge: Boolean): Unit
```

`dropTable` ...FIXME

Note	<code>dropTable</code> is used when: <ul style="list-style-type: none"> • <code>CreateViewCommand</code> logical command is executed • <code>DropTableCommand</code> logical command is executed • <code>DataStreamWriter</code> is requested to save a DataFrame to a table (with <code>Overwrite</code> save mode) • <code>CreateHiveTableAsSelectCommand</code> logical command is executed • <code>SessionCatalog</code> is requested to reset
------	---

- `CreateViewCommand` logical command is [executed](#)
- `DropTableCommand` logical command is [executed](#)
- `DataStreamWriter` is requested to [save a DataFrame to a table](#) (with `Overwrite` save mode)
- `CreateHiveTableAsSelectCommand` logical command is [executed](#)
- `SessionCatalog` is requested to [reset](#)

Getting Global Temporary View (Definition) — `getGlobalTempView` Method

```
getGlobalTempView(name: String): Option[LogicalPlan]
```

`getGlobalTempView` ...FIXME

Note	<code>getGlobalTempView</code> is used when...FIXME
------	---

`registerFunction` Method

```
registerFunction(  
    funcDefinition: CatalogFunction,  
    overrideIfExists: Boolean,  
    functionBuilder: Option[FunctionBuilder] = None): Unit
```

`registerFunction` ...FIXME

Note

- `registerFunction` is used when:
- `SessionCatalog` is requested to `lookupFunction`
 - `HiveSessionCatalog` is requested to `lookupFunction0`
 - `CreateFunctionCommand` logical command is executed

lookupFunctionInfo Method

```
lookupFunctionInfo(name: FunctionIdentifier): ExpressionInfo
```

```
lookupFunctionInfo ...FIXME
```

Note

`lookupFunctionInfo` is used when...FIXME

alterTableDataSchema Method

```
alterTableDataSchema(  
  identifier: TableIdentifier,  
  newDataSchema: StructType): Unit
```

```
alterTableDataSchema ...FIXME
```

Note

`alterTableDataSchema` is used when...FIXME

CatalogTable — Table Specification (Native Table Metadata)

`CatalogTable` is the **table specification**, i.e. the metadata of a table that is stored in a session-scoped catalog of relational entities (i.e. [SessionCatalog](#)).

```
scala> :type spark.sessionState.catalog
org.apache.spark.sql.catalyst.catalog.SessionCatalog

// Using high-level user-friendly catalog interface
scala> spark.catalog.listTables.filter($"name" === "t1").show
+-----+-----+-----+
|name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
| t1| default|      null|  MANAGED|    false|
+-----+-----+-----+-----+

// Using low-level internal SessionCatalog interface to access CatalogTables
val t1Tid = spark.sessionState.sqlParser.parseTableIdentifier("t1")
val t1Metadata = spark.sessionState.catalog.getTempViewOrPermanentTableMetadata(t1Tid)
scala> :type t1Metadata
org.apache.spark.sql.catalyst.catalog.CatalogTable
```

`CatalogTable` is [created](#) when:

- `SessionCatalog` is requested for a [table metadata](#)
- `HiveClientImpl` is requested for looking up a table in a metastore
- `DataFrameWriter` is requested to [create a table](#)
- `InsertIntoHiveDirCommand` logical command is executed
- `SparkSqlAstBuilder` does [visitCreateTable](#) and [visitCreateHiveTable](#)
- `CreateTableLikeCommand` logical command is executed
- `CreateViewCommand` logical command is [executed](#) (and [prepareTable](#))
- `CatalogImpl` is requested to [createTable](#)

The **readable text representation** of a `CatalogTable` (aka `simpleString`) is...FIXME

Note	<code>simpleString</code> is used exclusively when <code>ShowTablesCommand</code> logical command is executed (with a partition specification).
------	---

`CatalogTable` uses the following **text representation** (i.e. `toString`)...FIXME

`CatalogTable` is [created](#) with the optional [bucketing specification](#) that is used for the following:

- `CatalogImpl` is requested to [list the columns of a table](#)
- `FindDataSourceTable` logical evaluation rule is requested to [readDataSourceTable](#) (when [executed](#) for data source tables)
- `CreateTableLikeCommand` logical command is executed
- `DescribeTableCommand` logical command is requested to [describe detailed partition and storage information](#) (when [executed](#))
- `ShowCreateTableCommand` logical command is executed
- `CreateDataSourceTableCommand` and `CreateDataSourceTableAsSelectCommand` logical commands are executed
- `CatalogTable` is requested to [convert itself to LinkedHashMap](#)
- `HiveExternalCatalog` is requested to `doCreateTable`, `tableMetaToTableProps`, `doAlterTable`, `restoreHiveSerdeTable` and `restoreDataSourceTable`
- `HiveClientImpl` is requested to [retrieve a table metadata if available](#) and `toHiveTable`
- `InsertIntoHiveTable` logical command is [processInsert](#) (when [executed](#))
- `DataFrameWriter` is requested to [create a table](#) (via `saveAsTable`)
- `SparksqLAstBuilder` is requested to `visitCreateTable` and `visitCreateHiveTable`

Table Statistics for Query Planning (Auto Broadcast Joins and Cost-Based Optimization)

You manage a table metadata using the `catalog` interface (aka *metastore*). Among the management tasks is to get the [statistics](#) of a table (that are used for [cost-based query optimization](#)).

```
scala> t1Metadata.stats.foreach(println)
CatalogStatistics(714,Some(2),Map(p1 -> ColumnStat(2,Some(0),Some(1),0,4,4,None), id -> ColumnStat(2,Some(0),Some(1),0,4,4,None)))
 
scala> t1Metadata.stats.map(_.simpleString).foreach(println)
714 bytes, 2 rows
```

Note

The [CatalogStatistics](#) are optional when `CatalogTable` is [created](#).

Caution	FIXME When are stats specified? What if there are not?
---------	--

Unless [CatalogStatistics](#) are available in a table metadata (in a catalog) for a non-streaming [file data source table](#), [DataSource](#) [creates](#) a [HadoopFsRelation](#) with the table size specified by [spark.sql.defaultSizeInBytes](#) internal property (default: [Long.MaxValue](#)) for query planning of joins (and possibly to auto broadcast the table).

Internally, Spark alters table statistics using [ExternalCatalog.doAlterTableStats](#).

Unless [CatalogStatistics](#) are available in a table metadata (in a catalog) for [HiveTableRelation](#) (and [hive provider](#)) [DetermineTableStats](#) logical resolution rule can compute the table size using HDFS (if [spark.sql.statistics.fallBackToHdfs](#) property is turned on) or assume [spark.sql.defaultSizeInBytes](#) (that effectively disables table broadcasting).

When requested to [look up a table in a metastore](#), [HiveClientImpl](#) [reads table or partition statistics directly from a Hive metastore](#).

You can use [AnalyzeColumnCommand](#), [AnalyzePartitionCommand](#), [AnalyzeTableCommand](#) commands to record statistics in a catalog.

The table statistics can be [automatically updated](#) (after executing commands like [AlterTableAddPartitionCommand](#)) when [spark.sql.statistics.size.autoUpdate.enabled](#) property is turned on.

You can use [DESCRIBE](#) SQL command to show the histogram of a column if stored in a catalog.

dataSchema Method

```
dataSchema: StructType
```

[dataSchema ...FIXME](#)

Note	dataSchema is used when...FIXME
------	---

partitionSchema Method

```
partitionSchema: StructType
```

[partitionSchema ...FIXME](#)

Note	partitionSchema is used when...FIXME
------	--

Converting Table Specification to LinkedHashMap

— `toLinkedHashMap` Method

```
toLinkedHashMap: mutable.LinkedHashMap[String, String]
```

`toLinkedHashMap` converts the table specification to a collection of pairs (`LinkedHashMap[String, String]`) with the following fields and their values:

- **Database** with the database of the [TableIdentifier](#)
- **Table** with the table of the [TableIdentifier](#)
- **Owner** with the [owner](#) (if defined)
- **Created Time** with the [createTime](#)
- **Created By** with `spark` and the [createVersion](#)
- **Type** with the name of the [CatalogTableType](#)
- **Provider** with the [provider](#) (if defined)
- **Bucket specification** (of the [BucketSpec](#) if defined)
- **Comment** with the [comment](#) (if defined)
- **View Text, View Default Database and View Query Output Columns** for [VIEW](#) table type
- **Table Properties** with the [tableProperties](#) (if not empty)
- **Statistics** with the [CatalogStatistics](#) (if defined)
- **Storage specification** (of the [CatalogStorageFormat](#) if defined)
- **Partition Provider** with [Catalog](#) if the [tracksPartitionsInCatalog](#) flag is on
- **Partition Columns** with the [partitionColumns](#) (if not empty)
- **Schema** with the [schema](#) (if not empty)

Note	<p><code>toLinkedHashMap</code> is used when:</p> <ul style="list-style-type: none"> • <code>DescribeTableCommand</code> is requested to describeFormattedTableInfo (when <code>DescribeTableCommand</code> is requested to run for a non-temporary table and the isExtended flag on) • <code>CatalogTable</code> is requested for either a simple or a catalog text representation
-------------	---

Creating CatalogTable Instance

`CatalogTable` takes the following when created:

- `TableIdentifier`
- `CatalogTableType` (i.e. `EXTERNAL`, `MANAGED` or `VIEW`)
- [CatalogStorageFormat](#)
- [Schema](#)
- Name of the table provider (optional)
- Partition column names
- Optional [Bucketing specification](#) (default: `None`)
- Owner
- Create time
- Last access time
- Create version
- Properties
- Optional [table statistics](#)
- Optional view text
- Optional comment
- Unsupported features
- `tracksPartitionsInCatalog` flag
- `schemaPreservesCase` flag
- Ignored properties

database Method

```
database: String
```

`database` simply returns the database (of the [TableIdentifier](#)) or throws an `AnalysisException`:

```
table [identifier] did not specify database
```

Note	database is used when...FIXME
------	-------------------------------

CatalogStorageFormat — Storage Specification of Table or Partition

`CatalogStorageFormat` is the **storage specification** of a partition or a table, i.e. the metadata that includes the following:

- Location URI
- Input format
- Output format
- SerDe
- `compressed` flag
- Properties (as `Map[String, String]`)

`CatalogStorageFormat` is **created** when:

- `HiveClientImpl` is requested for metadata of a `table` or `table partition`
- `SparkSqlAstBuilder` is requested to parse Hive-specific `CREATE TABLE` or `INSERT OVERWRITE DIRECTORY` SQL statements

`CatalogStorageFormat` uses the following **text representation** (i.e. `toString`)...FIXME

Converting Storage Specification to LinkedHashMap — `toLinkedHashMap` Method

```
toLinkedHashMap: mutable.LinkedHashMap[String, String]
```

`toLinkedHashMap` ...FIXME

Note

- `toLinkedHashMap` is used when:
- `CatalogStorageFormat` is requested for a `text representation`
 - `CatalogTablePartition` is requested for `toLinkedHashMap`
 - `CatalogTable` is requested for `toLinkedHashMap`
 - `DescribeTableCommand` is requested to `run`

CatalogTablePartition — Partition Specification of Table

`CatalogTablePartition` is the **partition specification** of a table, i.e. the metadata of the partitions of a table.

`CatalogTablePartition` is **created** when:

- `HiveClientImpl` is requested to **retrieve a table partition metadata**
- `AlterTableAddPartitionCommand` and `AlterTableRecoverPartitionsCommand` logical commands are executed

`CatalogTablePartition` can hold the **table statistics** that...FIXME

The **readable text representation** of a `CatalogTablePartition` (aka `simpleString`) is...
FIXME

Note	<code>simpleString</code> is used exclusively when <code>ShowTablesCommand</code> is executed (with a partition specification).
------	--

`CatalogTablePartition` uses the following **text representation** (i.e. `toString`)...FIXME

Creating CatalogTablePartition Instance

`CatalogTablePartition` takes the following when created:

- Partition specification
- **CatalogStorageFormat**
- Parameters (default: an empty collection)
- **Table statistics** (default: `None`)

Converting Partition Specification to LinkedHashMap — `toLinkedHashMap` Method

```
toLinkedHashMap: mutable.LinkedHashMap[String, String]
```

`toLinkedHashMap` converts the partition specification to a collection of pairs (`LinkedHashMap[String, String]`) with the following fields and their values:

- **Partition Values** with the [spec](#)
- **Storage specification** (of the given [CatalogStorageFormat](#))
- **Partition Parameters** with the [parameters](#) (if not empty)
- **Partition Statistics** with the [CatalogStatistics](#) (if available)

Note

`toLinkedHashMap` is used when:

- `DescribeTableCommand` logical command is [executed](#) (with the [isExtended](#) flag on and a non-empty [partitionSpec](#)).
- `CatalogTablePartition` is requested for either a [simple](#) or a [catalog](#) text representation

location Method

`location: URI`

`location` simply returns the [location URI](#) of the [CatalogStorageFormat](#) or throws an [AnalysisException](#):

`Partition [[specString]] did not specify locationUri`

Note

`location` is used when...FIXME

BucketSpec — Bucketing Specification of Table

`BucketSpec` is the **bucketing specification** of a table, i.e. the metadata of the [bucketing](#) of a table.

`BucketSpec` includes the following:

- Number of buckets
- Bucket column names - the names of the columns used for buckets (at least one)
- Sort column names - the names of the columns used to sort data in buckets

The [number of buckets](#) has to be between `0` and `100000` exclusive (or an `AnalysisException` is thrown).

`BucketSpec` is [created](#) when:

1. `DataFrameWriter` is requested to [saveAsTable](#) (and does `getBucketSpec`)
2. `HiveExternalCatalog` is requested to [getBucketSpecFromTableProperties](#) and [tableMetaToTableProps](#)
3. `HiveClientImpl` is requested to [retrieve a table metadata](#)
4. `SparkSqlAstBuilder` is requested to [visitBucketSpec](#) (for `CREATE TABLE` SQL statement with `CLUSTERED BY` and `INTO n BUCKETS` with optional `SORTED BY` clauses)

`BucketSpec` uses the following **text representation** (i.e. `toString`):

```
[numBuckets] buckets, bucket columns: [[bucketColumnNames]], sort columns: [[sortColumnNames]]
```

```
import org.apache.spark.sql.catalyst.catalog.BucketSpec
val bucketSpec = BucketSpec(
  numBuckets = 8,
  bucketColumnNames = Seq("col1"),
  sortColumnNames = Seq("col2"))
scala> println(bucketSpec)
8 buckets, bucket columns: [col1], sort columns: [col2]
```

Converting Bucketing Specification to LinkedHashMap — `toLinkedHashMap` Method

```
toLinkedHashMap: mutable.LinkedHashMap[String, String]
```

`toLinkedHashMap` converts the bucketing specification to a collection of pairs (`LinkedHashMap[String, String]`) with the following fields and their values:

- **Num Buckets** with the [numBuckets](#)
- **Bucket Columns** with the [bucketColumnNames](#)
- **Sort Columns** with the [sortColumnNames](#)

`toLinkedHashMap` quotes the column names.

```
scala> println(bucketSpec.toLinkedHashMap)
Map(Num Buckets -> 8, Bucket Columns -> [`col1`], Sort Columns -> [`col2`])
```

Note

`toLinkedHashMap` is used when:

- `CatalogTable` is requested for [toLinkedHashMap](#)
- `DescribeTableCommand` logical command is [executed](#) with a non-empty `partitionSpec` and the `isExtended` flag on (that uses `describeFormattedDetailedPartitionInfo`).

HiveSessionCatalog — Hive-Specific Catalog of Relational Entities

`HiveSessionCatalog` is a session-scoped catalog of relational entities that is used when `SparkSession` was created with [Hive support enabled](#).

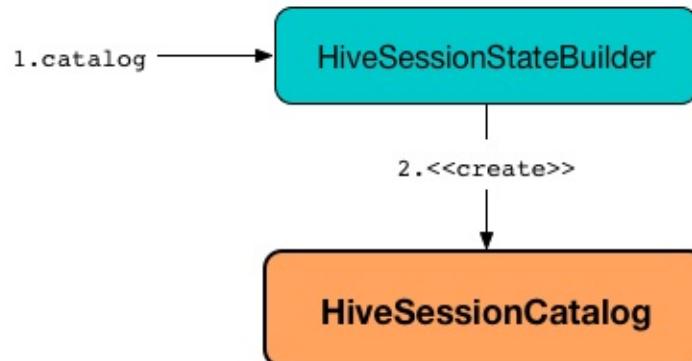


Figure 1. `HiveSessionCatalog` and `HiveSessionStateBuilder`

`HiveSessionCatalog` is available as `catalog` property of `SessionState` when `SparkSession` was created with [Hive support enabled](#) (that in the end sets `spark.sql.catalogImplementation` internal configuration property to `hive`).

```

import org.apache.spark.sql.internal.StaticSQLConf
val catalogType = spark.conf.get(StaticSQLConf.CATALOG_IMPLEMENTATION.key)
scala> println(catalogType)
hive

// You could also use the property key by name
scala> spark.conf.get("spark.sql.catalogImplementation")
res1: String = hive

// Since Hive is enabled HiveSessionCatalog is the implementation
scala> spark.sessionState.catalog
res2: org.apache.spark.sql.catalyst.catalog.SessionCatalog = org.apache.spark.sql.hive.
HiveSessionCatalog@1ae3d0a8
  
```

`HiveSessionCatalog` is [created](#) exclusively when `HiveSessionStateBuilder` is requested for the `SessionCatalog`.

`HiveSessionCatalog` uses the legacy `HiveMetastoreCatalog` (which is another session-scoped catalog of relational entities) exclusively to allow `RelationConversions` logical evaluation rule to [convert Hive metastore relations to data source relations](#) when [executed](#).

Creating `HiveSessionCatalog` Instance

`HiveSessionCatalog` takes the following when created:

- `HiveExternalCatalog`
- `GlobalTempViewManager`
- Legacy `HiveMetastoreCatalog`
- `FunctionRegistry`
- `SQLConf`
- Hadoop `Configuration`
- `ParserInterface`
- `FunctionResourceLoader`

lookupFunction0 Internal Method

```
lookupFunction0(name: FunctionIdentifier, children: Seq[Expression]): Expression
```

`lookupFunction0` ...FIXME

Note	<code>lookupFunction0</code> is used when...FIXME
------	---

HiveMetastoreCatalog — Legacy SessionCatalog for Converting Hive Metastore Relations to Data Source Relations

`HiveMetastoreCatalog` is a legacy [session-scoped catalog](#) of relational entities that `HiveSessionCatalog` [uses](#) exclusively for [converting](#) Hive metastore relations to data source [relations](#) (when `RelationConversions` logical evaluation rule is [executed](#)).

`HiveMetastoreCatalog` is [created](#) exclusively when `HiveSessionStateBuilder` is requested for `SessionCatalog` (and creates a `HiveSessionCatalog`).

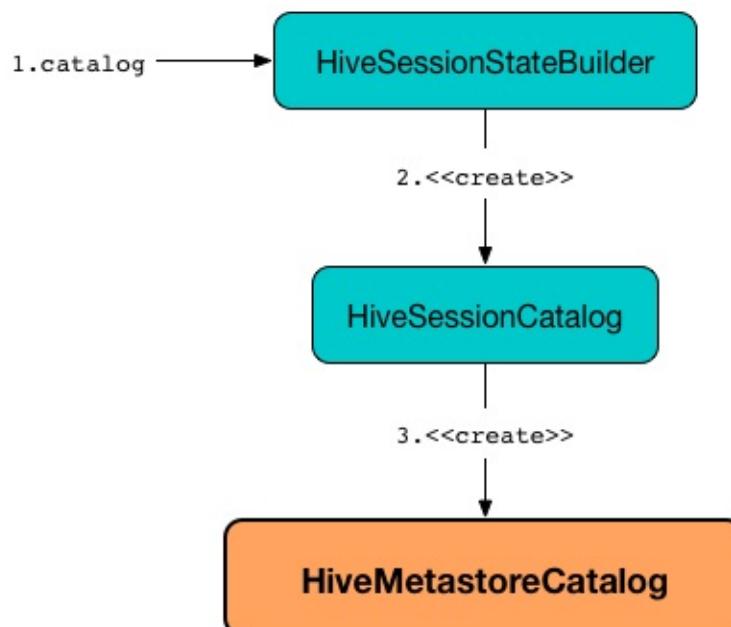


Figure 1. `HiveMetastoreCatalog`, `HiveSessionCatalog` and `HiveSessionStateBuilder` `HiveMetastoreCatalog` takes a [SparkSession](#) when created.

Converting HiveTableRelation to LogicalRelation — `convertToLogicalRelation` Method

```
convertToLogicalRelation(  
    relation: HiveTableRelation,  
    options: Map[String, String],  
    fileFormatClass: Class[_ <: FileFormat],  
    fileType: String): LogicalRelation
```

`convertToLogicalRelation` ...FIXME

Note

`convertToLogicalRelation` is used exclusively when `RelationConversions` logical evaluation rule is requested to [convert a HiveTableRelation to a LogicalRelation](#) for `parquet`, `native` and `hive` ORC storage formats.

inferIfNeeded Internal Method

```
inferIfNeeded(  
    relation: HiveTableRelation,  
    options: Map[String, String],  
    fileFormat: FileFormat,  
    fileIndexOpt: Option[FileIndex] = None): CatalogTable
```

`inferIfNeeded` ...FIXME

Note

`inferIfNeeded` is used exclusively when `HiveMetastoreCatalog` is requested to [convertToLogicalRelation](#).

SessionState — State Separation Layer Between SparkSessions

`SessionState` is the [state separation layer](#) between Spark SQL sessions, including SQL configuration, tables, functions, UDFs, SQL parser, and everything else that depends on a [SQLConf](#).

`SessionState` is available as the [sessionState](#) property of a `SparkSession`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState
org.apache.spark.sql.internal.SessionState
```

`SessionState` is [created](#) when `SparkSession` is requested to [instantiateSessionState](#) (when requested for the `SessionState` per `spark.sql.catalogImplementation` configuration property).

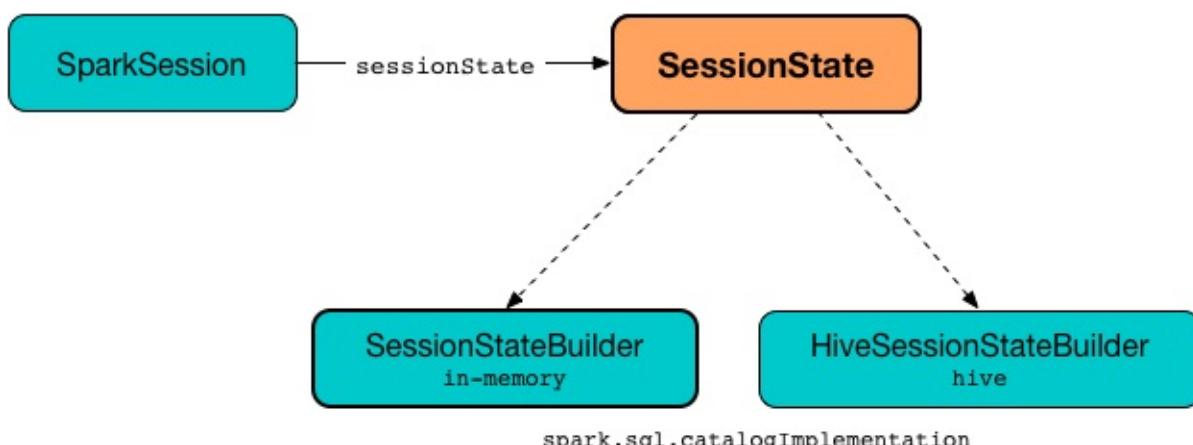


Figure 1. Creating SessionState

Note

When requested for the `SessionState`, `SparkSession` uses `spark.sql.catalogImplementation` configuration property to load and create a `BaseSessionStateBuilder` that is then requested to [create a SessionState instance](#).

There are two `BaseSessionStateBuilders` available:

- (default) `SessionStateBuilder` for `in-memory` catalog
- `HiveSessionStateBuilder` for `hive` catalog

`hive` catalog is set when the `SparkSession` was [created](#) with the Hive support enabled (using `Builder.enableHiveSupport`).

Table 1. SessionState's (Lazily-Initialized) Attributes

Name	Type	Description
analyzer	Analyzer	<p>Spark Analyzer</p> <p>Initialized lazily (i.e. only when requested the first time) using the analyzerBuilder factory function.</p> <p>Used when...FIXME</p>
catalog	SessionCatalog	<p>Metastore of tables and databases</p> <p>Used when...FIXME</p>
conf	SQLConf	<p>FIXME</p> <p>Used when...FIXME</p>
experimentalMethods	ExperimentalMethods	<p>FIXME</p> <p>Used when...FIXME</p>
functionRegistry	FunctionRegistry	<p>FIXME</p> <p>Used when...FIXME</p>
listenerManager	ExecutionListenerManager	<p>FIXME</p> <p>Used when...FIXME</p>
optimizer	Optimizer	<p>Logical query plan optimizer</p> <p>Used exclusively when QueryExecution creates an optimized logical plan.</p>
resourceLoader	SessionResourceLoader	<p>FIXME</p> <p>Used when...FIXME</p>
sqlParser	ParserInterface	<p>FIXME</p> <p>Used when...FIXME</p>
streamingQueryManager	StreamingQueryManager	<p>Used to manage streaming queries in Spark Structured Streaming</p>
udfRegistration	UDFRegistration	<p>Interface to register user-defined functions.</p>

Used when...FIXME

Note

`SessionState` is a `private[sql]` class and, given the package `org.apache.spark.sql.internal`, `SessionState` should be considered *internal*.

Creating SessionState Instance

`SessionState` takes the following when created:

- `SharedState`
- `SQLConf`
- `ExperimentalMethods`
- `FunctionRegistry`
- `UDFRegistration`
- `catalogBuilder` function to create a `SessionCatalog` (i.e. `() => SessionCatalog`)
- `ParserInterface`
- `analyzerBuilder` function to create an `Analyzer` (i.e. `() => Analyzer`)
- `optimizerBuilder` function to create an `Optimizer` (i.e. `() => Optimizer`)
- `SparkPlanner`
- Spark Structured Streaming's `streamingQueryManager`
- `ExecutionListenerManager`
- `resourceLoaderBuilder` function to create a `SessionResourceLoader` (i.e. `() => SessionResourceLoader`)
- `createQueryExecution` function to create a `QueryExecution` given a `LogicalPlan` (i.e. `LogicalPlan => QueryExecution`)
- `createClone` function to clone the `SessionState` given a `SparkSession` (i.e. `(SparkSession, SessionState) => SessionState`)

clone Method

```
clone(newSparkSession: SparkSession): SessionState
```

`clone` ...FIXME

Note	<code>clone</code> is used when...
------	------------------------------------

"Executing" Logical Plan (Creating QueryExecution For LogicalPlan) — `executePlan` Method

```
executePlan(plan: LogicalPlan): QueryExecution
```

`executePlan` simply executes the `createQueryExecution` function on the input logical plan (that simply creates a `QueryExecution` with the current `SparkSession` and the input logical plan).

refreshTable Method

```
refreshTable(tableName: String): Unit
```

```
refreshTable ...FIXME
```

Note	<code>refreshTable</code> is used...FIXME
------	---

Creating New Hadoop Configuration — `newHadoopConf` Method

```
newHadoopConf(): Configuration
```

`newHadoopConf` returns a Hadoop `Configuration` (with the `SparkContext.hadoopConfiguration` and all the configuration properties of the `SQLConf`).

Note	<code>newHadoopConf</code> is used by <code>ScriptTransformation</code> , <code>ParquetRelation</code> , <code>StateStoreRDD</code> , and <code>SessionState</code> itself, and few other places.
------	---

Creating New Hadoop Configuration With Extra Options — `newHadoopConfWithOptions` Method

```
newHadoopConfWithOptions(options: Map[String, String]): Configuration
```

`newHadoopConfWithOptions` creates a new Hadoop Configuration with the input `options` set (except `path` and `paths` options that are skipped).

	<p><code>newHadoopConfWithOptions</code> is used when:</p> <ul style="list-style-type: none">• <code>TextBasedFileFormat</code> is requested to say whether it is splitable or not• <code>FileSourceScanExec</code> is requested for the input RDD• <code>InsertIntoHadoopFsRelationCommand</code> is requested to run• <code>PartitioningAwareFileIndex</code> is requested for the Hadoop Configuration
Note	

BaseSessionStateBuilder — Generic Builder of SessionState

`BaseSessionStateBuilder` is the [contract](#) of **builder objects** that coordinate construction of a new [SessionState](#).

Table 1. BaseSessionStateBuilders

BaseSessionStateBuilder	Description
SessionStateBuilder	
HiveSessionStateBuilder	

`BaseSessionStateBuilder` is [created](#) when `SparkSession` is requested for a [SessionState](#).

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState
org.apache.spark.sql.internal.SessionState
```

`BaseSessionStateBuilder` requires that [implementations](#) define `newBuilder` method that `SparkSession` uses (indirectly) when requested for the [SessionState](#) (per `spark.sql.catalogImplementation` internal configuration property).

```
newBuilder: (SparkSession, Option[SessionState]) => BaseSessionStateBuilder
```

Note	<code>BaseSessionStateBuilder</code> and <code>spark.sql.catalogImplementation</code> configuration property allow for Hive and non-Hive Spark deployments.
------	---

`BaseSessionStateBuilder` holds [properties](#) that (together with `newBuilder`) are used to create a [SessionState](#).

Table 2. BaseSessionStateBuilder's Properties

Name	Description
<code>analyzer</code>	<code>Logical analyzer</code>
<code>catalog</code>	<code>SessionCatalog</code> Used to create <code>Analyzer</code> , <code>Optimizer</code> and a <code>SessionState</code> itself

	Note	HiveSessionStateBuilder manages its own Hive-aware HiveSessionCatalog .
<code>conf</code>		SQLConf
<code>customOperatorOptimizationRules</code>		Custom operator optimization rules to add to the base Operator Optimization batch . When requested for the custom rules, <code>customOperatorOptimizationRules</code> simply requests the SparkSessionExtensions to <code>buildOptimizerRules</code> .
<code>experimentalMethods</code>		ExperimentalMethods
<code>extensions</code>		SparkSessionExtensions
<code>functionRegistry</code>		FunctionRegistry
<code>listenerManager</code>		ExecutionListenerManager
<code>optimizer</code>		SparkOptimizer (that is downcast to the base Optimizer) that is created with the SessionCatalog and the ExperimentalMethods . Note that the <code>SparkOptimizer</code> adds the <code>customOperatorOptimizationRules</code> to the <code>operator</code> optimization rules. <code>optimizer</code> is used when <code>BaseSessionStateBuilder</code> is requested to create a SessionState (for the <code>optimizerBuilder</code> function to create an Optimizer when requested for the Optimizer).
<code>planner</code>		SparkPlanner
<code>resourceLoader</code>		SessionResourceLoader
<code>sqlParser</code>		ParserInterface
<code>streamingQueryManager</code>		Spark Structured Streaming's StreamingQueryManager
<code>udfRegistration</code>		UDFRegistration

Note

`BaseSessionStateBuilder` defines a type alias `NewBuilder` for a function to create a `BaseSessionStateBuilder`.

```
type NewBuilder = (SparkSession, Option[SessionState]) => BaseSessionStateBuilder
```

Note

`BaseSessionStateBuilder` is an experimental and unstable API.

Creating Function to Build SessionState — `createClone` Method

```
createClone: (SparkSession, SessionState) => SessionState
```

`createClone` gives a function of `SparkSession` and `SessionState` that executes `newBuilder` followed by `build`.

Note

`createClone` is used exclusively when `BaseSessionStateBuilder` is requested for a `SessionState`

Creating SessionState Instance — `build` Method

```
build(): SessionState
```

`build` creates a `SessionState` with the following:

- `SharedState` of `SparkSession`
- `SQLConf`
- `ExperimentalMethods`
- `FunctionRegistry`
- `UDFRegistration`
- `SessionCatalog`
- `ParserInterface`
- `Analyzer`
- `Optimizer`
- `SparkPlanner`

- [StreamingQueryManager](#)
- [ExecutionListenerManager](#)
- [SessionResourceLoader](#)
- [createQueryExecution](#)
- [createClone](#)

Note`build` is used when:

- `SparkSession` is requested for the `SessionState` (and `builds it using a class name` per `spark.sql.catalogImplementation` configuration property)
- `BaseSessionStateBuilder` is requested to `create a clone` of a `SessionState`

Creating BaseSessionStateBuilder Instance

`BaseSessionStateBuilder` takes the following when created:

- [SparkSession](#)
- Optional [SessionState](#)

Getting Function to Create QueryExecution For LogicalPlan — `createQueryExecution` Method

```
createQueryExecution: LogicalPlan => QueryExecution
```

`createQueryExecution` simply returns a function that takes a `LogicalPlan` and creates a `QueryExecution` with the `SparkSession` and the logical plan.

Note

`createQueryExecution` is used exclusively when `BaseSessionStateBuilder` is requested to `create a SessionState instance`.

SessionStateBuilder

SessionStateBuilder is...FIXME

HiveSessionStateBuilder — Builder of Hive-Specific SessionState

`HiveSessionStateBuilder` is a `BaseSessionStateBuilder` that has Hive-specific `Analyzer`, `SparkPlanner`, `HiveSessionCatalog`, `HiveExternalCatalog` and `HiveSessionResourceLoader`.

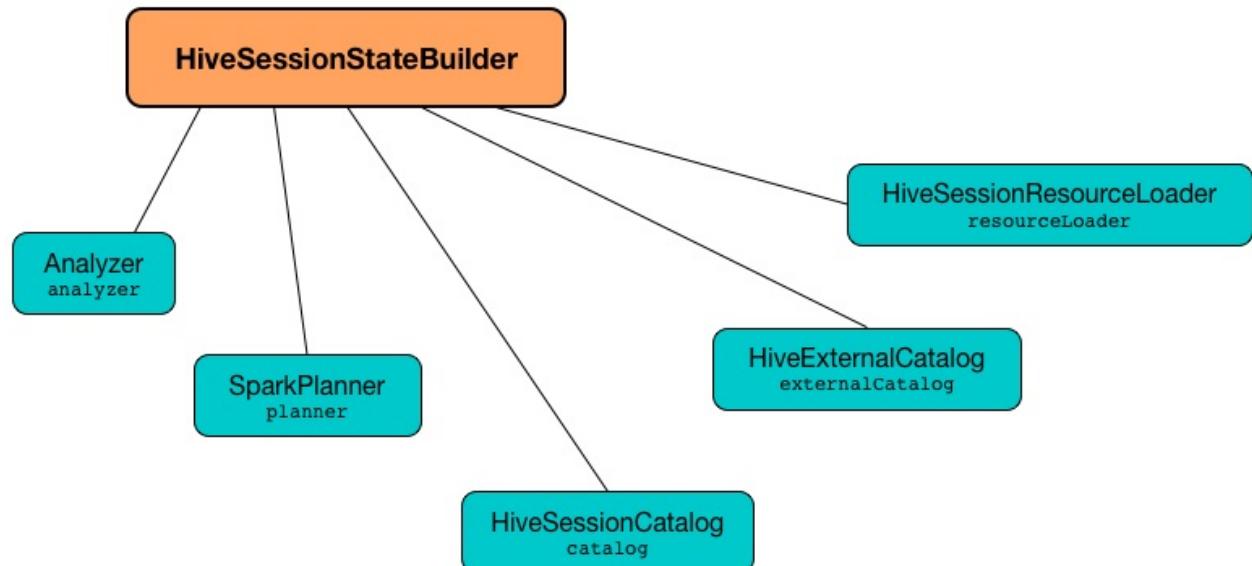


Figure 1. `HiveSessionStateBuilder`'s Hive-Specific Properties

`HiveSessionStateBuilder` is created (using `newBuilder`) exclusively when...FIXME

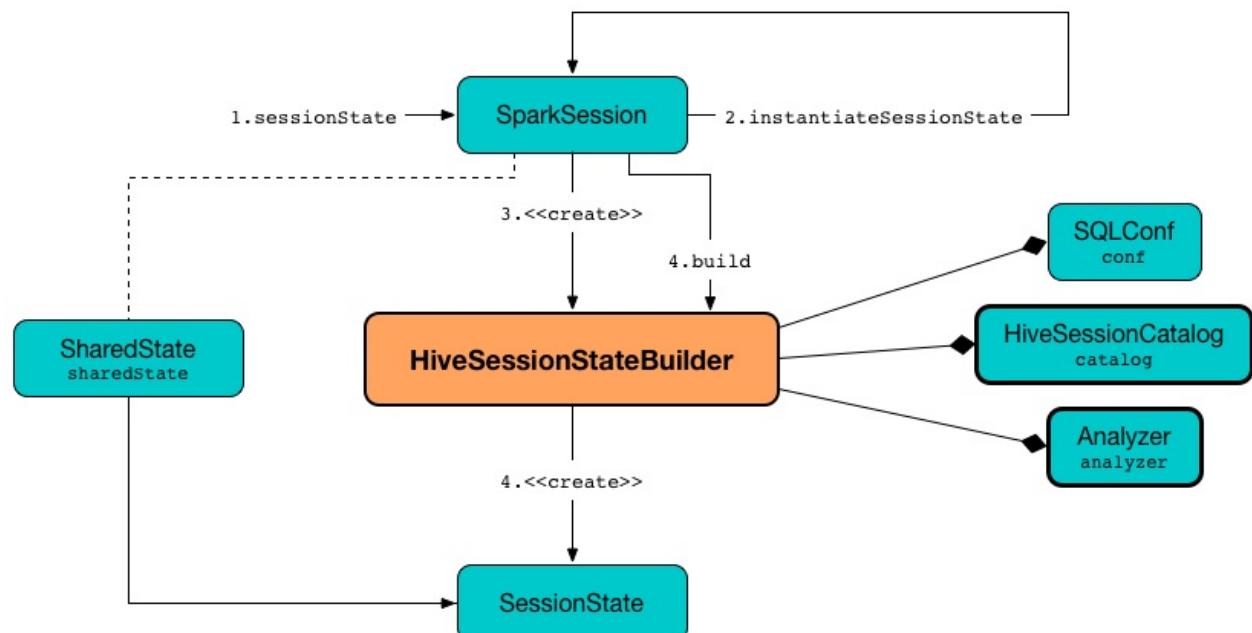


Figure 2. `HiveSessionStateBuilder` and `SessionState` (in `SparkSession`)

Table 1. HiveSessionStateBuilder's Properties

Name	Description		
analyzer	Hive-specific logical query plan analyzer with the Hive-specific rules .		
catalog	HiveSessionCatalog with the following: <ul style="list-style-type: none"> • HiveExternalCatalog • GlobalTempViewManager from the session-specific SharedState • New HiveMetastoreCatalog • FunctionRegistry • SQLConf • New Hadoop Configuration • ParserInterface • HiveSessionResourceLoader <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">Note</td> <td style="padding: 2px;">If <code>parentState</code> is defined, the state is copied to <code>catalog</code></td> </tr> </table> </div>	Note	If <code>parentState</code> is defined, the state is copied to <code>catalog</code>
Note	If <code>parentState</code> is defined, the state is copied to <code>catalog</code>		
	Used to create Hive-specific Analyzer and a RelationConversions logical evaluation rule (as part of Hive-Specific Analyzer's PostHoc Resolution Rules)		
externalCatalog	HiveExternalCatalog		
planner	SparkPlanner with Hive-specific strategies .		
resourceLoader	HiveSessionResourceLoader		

SparkPlanner with Hive-Specific Strategies — `planner` Property

```
planner: SparkPlanner
```

Note	<code>planner</code> is part of BaseSessionStateBuilder Contract to create a query planner.
------	---

`planner` is a [SparkPlanner](#) with...FIXME

`planner` uses the [Hive-specific strategies](#).

Table 2. Hive-Specific SparkPlanner's Hive-Specific Strategies

Strategy	Description
HiveTableScans	
Scripts	

Logical Query Plan Analyzer with Hive-Specific Rules

— analyzer Property

analyzer: Analyzer

Note analyzer is part of [BaseSessionStateBuilder Contract](#) to create a logical query plan analyzer.

analyzer is a [Analyzer](#) with [Hive-specific SessionCatalog](#) (and [SQLConf](#)).

analyzer uses the Hive-specific [extended resolution](#), [postHoc resolution](#) and [extended check](#) rules.

Table 3. Hive-Specific Analyzer's Extended Resolution Rules (in the order of execution)

Logical Rule	Description
ResolveHiveSerdeTable	
FindDataSourceTable	
ResolveSQLOnFile	

Table 4. Hive-Specific Analyzer's PostHoc Resolution Rules

Logical Rule	Description
DetermineTableStats	
RelationConversions	
PreprocessTableCreation	
PreprocessTableInsertion	
DataSourceAnalysis	
HiveAnalysis	

Table 5. Hive-Specific Analyzer's Extended Check Rules

Logical Rule	Description
PreWriteCheck	
PreReadCheck	

Builder Function to Create HiveSessionStateBuilder — newBuilder Factory Method

```
newBuilder: NewBuilder
```

Note

`newBuilder` is part of [BaseSessionStateBuilder Contract](#) to...FIXME.

`newBuilder ...FIXME`

Creating HiveSessionStateBuilder Instance

`HiveSessionStateBuilder` takes the following when created:

- [SparkSession](#)
- Optional [SessionState](#) (`None` by default)

SharedState — State Shared Across SparkSessions

`SharedState` holds the [shared state](#) across multiple [SparkSessions](#).

Table 1. SharedState's Properties

Name	Type	Description
<code>cacheManager</code>	CacheManager	
<code>externalCatalog</code>	ExternalCatalog	Metastore of permanent relational entities, i.e. databases, tables, partitions and functions.
		<p>Note <code>externalCatalog</code> is initialized on the first access.</p>
<code>globalTempViewManager</code>	GlobalTempViewManager	Management interface for global temporary views
<code>jarClassLoader</code>	NonClosableMutableURLClassLoader	
<code>sparkContext</code>	SparkContext	Spark Core's <code>SparkContext</code>
<code>statusStore</code>	SQLAppStatusStore	
<code>warehousePath</code>	<code>String</code>	Warehouse path

`SharedState` is available as the [sharedState](#) property of a `SparkSession`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sharedState
org.apache.spark.sql.internal.SharedState
```

`SharedState` is shared across `SparkSessions`.

```
scala> spark.newSession.sharedState == spark.sharedState
res1: Boolean = true
```

`SharedState` is created exclusively when accessed using [sharedState](#) property of `SparkSession`.

Tip

Enable `INFO` logging level for `org.apache.spark.sql.internal.SharedState` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.internal.SharedState=INFO
```

Refer to [Logging](#).

warehousePath Property

`warehousePath: String`

`warehousePath` is the warehouse path with the value of:

1. `hive.metastore.warehouse.dir` if defined and `spark.sql.warehouse.dir` is not
2. `spark.sql.warehouse.dir` if `hive.metastore.warehouse.dir` is undefined

You should see the following INFO message in the logs when `SharedState` is created:

```
INFO Warehouse path is '[warehousePath]'.
```

`warehousePath` is used exclusively when `SharedState` initializes [ExternalCatalog](#) (and creates the default database in the metastore).

While initialized, `warehousePath` does the following:

1. Loads `hive-site.xml` if available on CLASSPATH, i.e. adds it as a configuration resource to Hadoop's [Configuration](#) (of `SparkContext`).
2. Removes `hive.metastore.warehouse.dir` from `SparkConf` (of `SparkContext`) and leaves it off if defined using any of the Hadoop configuration resources.
3. Sets `spark.sql.warehouse.dir` or `hive.metastore.warehouse.dir` in the Hadoop configuration (of `SparkContext`)
 - i. If `hive.metastore.warehouse.dir` has been defined in any of the Hadoop configuration resources but `spark.sql.warehouse.dir` has not, `spark.sql.warehouse.dir` becomes the value of `hive.metastore.warehouse.dir`.

You should see the following INFO message in the logs:

```
spark.sql.warehouse.dir is not set, but hive.metastore.warehouse.dir is set.
Setting spark.sql.warehouse.dir to the value of hive.metastore.warehouse.dir
(['[hiveWarehouseDir]']).
```

- ii. Otherwise, the Hadoop configuration's `hive.metastore.warehouse.dir` is set to `spark.sql.warehouse.dir`

You should see the following INFO message in the logs:

```
Setting hive.metastore.warehouse.dir ('[hiveWarehouseDir]') to the value of s
park.sql.warehouse.dir ('[sparkWarehouseDir]').
```

externalCatalog Property

```
externalCatalog: ExternalCatalog
```

`externalCatalog` is created reflectively per [spark.sql.catalogImplementation](#) internal configuration property (with the current Hadoop's [Configuration](#) as `SparkContext.hadoopConfiguration`):

- [HiveExternalCatalog](#) for `hive`
- [InMemoryCatalog](#) for `in-memory` (default)

While initialized:

1. Creates the **default** database (with `default database` description and `warehousePath` location) if [it doesn't exist](#).
2. Registers a `ExternalCatalogEventListener` that propagates external catalog events to the Spark listener bus.

externalCatalogClassName Internal Method

```
externalCatalogClassName(conf: SparkConf): String
```

`externalCatalogClassName` gives the name of the class of the [ExternalCatalog](#) per [spark.sql.catalogImplementation](#), i.e.

- [org.apache.spark.sql.hive.HiveExternalCatalog](#) for `hive`
- [org.apache.spark.sql.catalyst.catalog.InMemoryCatalog](#) for `in-memory`

Note

`externalCatalogClassName` is used exclusively when `SharedState` is requested for the [ExternalCatalog](#).

Accessing Management Interface of Global Temporary Views — `globalTempViewManager` Property

```
globalTempViewManager: GlobalTempViewManager
```

When accessed for the very first time, `globalTempViewManager` gets the name of the global temporary view database (as the value of `spark.sql.globalTempDatabase` internal static configuration property).

In the end, `globalTempViewManager` creates a new [GlobalTempViewManager](#) (with the database name).

`globalTempViewManager` throws a `SparkException` when the global temporary view database exist in the [ExternalCatalog](#).

[`globalTempDB`] is a system preserved database, please rename your existing database to resolve the name conflict, or set a different value for `spark.sql.globalTempDatabase`, and launch your Spark application again.

Note

`globalTempViewManager` is used when [BaseSessionStateBuilder](#) and [HiveSessionStateBuilder](#) are requested for the [SessionCatalog](#).

CacheManager—In-Memory Cache for Tables and Views

`CacheManager` is an in-memory cache (*registry*) for structured queries (by their logical plans).

`CacheManager` is shared across `SparkSessions` through `SharedState`.

```
val spark: SparkSession = ...
spark.sharedState.cacheManager
```

Note	A Spark developer can use <code>CacheManager</code> to cache <code>Dataset</code> s using <code>cache</code> or <code>persist</code> operators.
------	---

`CacheManager` uses the `cachedData` internal registry to manage cached structured queries and their `InMemoryRelation` cached representation.

`CacheManager` can be empty.

`CacheManager` uses `CachedData` data structure for managing cached structured queries with the `LogicalPlan` (of a structured query) and a corresponding `InMemoryRelation` leaf logical operator.

Tip	<p>Enable <code>ALL</code> logging level for <code>org.apache.spark.sql.execution.CacheManager</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.CacheManager=ALL</pre> <p>Refer to Logging.</p>
-----	---

Cached Structured Queries—`cachedData` Internal Registry

```
cachedData: LinkedList[CachedData]
```

`cachedData` is a collection of `CachedData`.

A new `CachedData` added when `CacheManager` is requested to:

- `cacheQuery`

- `recacheByCondition`

A `CachedData` removed when `CacheManager` is requested to:

- `uncacheQuery`
- `recacheByCondition`

All `CachedData` removed (cleared) when `CacheManager` is requested to `clearCache`

lookupCachedData Method

```
lookupCachedData(query: Dataset[_]): Option[CachedData]
lookupCachedData(plan: LogicalPlan): Option[CachedData]
```

`lookupCachedData ...FIXME`

Note

`lookupCachedData` is used when:

- `Dataset.storageLevel` basic action is used
- `CatalogImpl` is requested to `isCached`
- `CacheManager` is requested to `cacheQuery` and `useCachedData`

Un-caching Dataset— uncacheQuery Method

```
uncacheQuery(
  query: Dataset[_],
  cascade: Boolean,
  blocking: Boolean = true): Unit
uncacheQuery(
  spark: SparkSession,
  plan: LogicalPlan,
  cascade: Boolean,
  blocking: Boolean): Unit
```

`uncacheQuery ...FIXME`

Note

- `uncacheQuery` is used when:
- `Dataset.unpersist` basic action is used
 - `DropTableCommand` and `TruncateTableCommand` logical commands are executed
 - `CatalogImpl` is requested to `uncache` and `refresh` a table or view, `dropTempView` and `dropGlobalTempView`

isEmpty Method

```
isEmpty: Boolean
```

`isEmpty` simply says whether there are any `CachedData` entries in the `cachedData` internal registry.

Caching Dataset— cacheQuery Method

```
cacheQuery(  
    query: Dataset[_],  
    tableName: Option[String] = None,  
    storageLevel: StorageLevel = MEMORY_AND_DISK): Unit
```

`cacheQuery` adds the `analyzed logical plan` of the input `Dataset` to the `cachedData` internal registry of cached queries.

Internally, `cacheQuery` requests the `Dataset` for the `analyzed logical plan` and creates a `InMemoryRelation` with the following properties:

- `spark.sql.inMemoryColumnarStorage.compressed` (enabled by default)
- `spark.sql.inMemoryColumnarStorage.batchSize` (default: `10000`)
- Input `storageLevel` storage level (default: `MEMORY_AND_DISK`)
- `Optimized physical query plan` (after requesting `SessionState` to execute the analyzed logical plan)
- Input `tableName`
- `Statistics` of the analyzed query plan

`cacheQuery` then creates a `CachedData` (for the analyzed query plan and the `InMemoryRelation`) and adds it to the `cachedData` internal registry.

If the input `query` has already been cached, `cacheQuery` simply prints the following WARN message to the logs and exits (i.e. does nothing but prints out the WARN message):

```
Asked to cache already cached data.
```

Note

`cacheQuery` is used when:

- `Dataset.persist` basic action is used
- `CatalogImpl` is requested to `cache` and `refresh` a table or view in-memory

Removing All Cached Logical Plans — `clearCache` Method

```
clearCache(): Unit
```

`clearCache` takes every `CachedData` from the `cachedData` internal registry and requests it for the `InMemoryRelation` to access the `CachedRDDBuilder`. `clearCache` requests the `CachedRDDBuilder` to `clearCache`.

In the end, `clearCache` removes all `CachedData` entries from the `cachedData` internal registry.

Note

`clearCache` is used exclusively when `CatalogImpl` is requested to `clear the cache`.

Re-Caching Structured Query — `recacheByCondition` Internal Method

```
recacheByCondition(spark: SparkSession, condition: LogicalPlan => Boolean): Unit
```

```
recacheByCondition ...FIXME
```

Note

`recacheByCondition` is used when `CacheManager` is requested to `uncache a structured query, recacheByPlan, and recacheByPath`.

`recacheByPlan` Method

```
recacheByPlan(spark: SparkSession, plan: LogicalPlan): Unit
```

```
recacheByPlan ...FIXME
```

Note	recacheByPlan is used exclusively when <code>InsertIntoDataSourceCommand</code> logical command is executed .
------	---

recacheByPath Method

```
recacheByPath(spark: SparkSession, resourcePath: String): Unit
```

```
recacheByPath ...FIXME
```

Note	recacheByPath is used exclusively when <code>CatalogImpl</code> is requested to refreshByPath .
------	---

Replacing Segments of Logical Query Plan With Cached Data — useCachedData Method

```
useCachedData(plan: LogicalPlan): LogicalPlan
```

```
useCachedData ...FIXME
```

Note	useCachedData is used exclusively when <code>queryExecution</code> is requested for a cached logical query plan .
------	---

lookupAndRefresh Internal Method

```
lookupAndRefresh(  
    plan: LogicalPlan,  
    fs: FileSystem,  
    qualifiedPath: Path): Boolean
```

```
lookupAndRefresh ...FIXME
```

Note	lookupAndRefresh is used exclusively when <code>CacheManager</code> is requested to recacheByPath .
------	---

CachedRDDBuilder

`CachedRDDBuilder` is created exclusively when `InMemoryRelation` leaf logical operator is created.

`CachedRDDBuilder` uses a `RDD` of `CachedBatches` that is either given or built internally.

`CachedRDDBuilder` uses `CachedBatch` data structure with the following attributes:

- Number of rows
- Buffers (`Array[Array[Byte]]`)
- Statistics (`InternalRow`)

`CachedRDDBuilder` uses `isCachedColumnBuffersLoaded` flag that is enabled (`true`) when the `_cachedColumnBuffers` is defined (not `null`). `isCachedColumnBuffersLoaded` is used exclusively when `CacheManager` is requested to `recacheByCondition`.

`CachedRDDBuilder` uses `sizeInBytesStats` metric (`LongAccumulator`) to `buildBuffers` and when `InMemoryRelation` is requested to `computeStats`.

Creating CachedRDDBuilder Instance

`CachedRDDBuilder` takes the following to be created:

- `useCompression` flag
- Batch size
- `StorageLevel`
- Physical operator
- Table name
- `RDD[CachedBatch]` (default: `null`)

`CachedRDDBuilder` initializes the internal registries and counters.

buildBuffers Internal Method

```
buildBuffers(): RDD[CachedBatch]
```

`buildBuffers` ...FIXME

Note

`buildBuffers` is used exclusively when `CachedRDDBuilder` is requested to [cachedColumnBuffers](#).

clearCache Method

```
clearCache(blocking: Boolean = true): Unit
```

clearCache ...FIXME

Note

`clearCache` is used exclusively when `CacheManager` is requested to [clearCache](#), [uncacheQuery](#), and [recacheByCondition](#).

RuntimeConfig — Management Interface of Runtime Configuration

`RuntimeConfig` is the [management interface](#) of the [runtime configuration](#).

Table 1. RuntimeConfig API

Method	Description
<code>get</code>	<code>get(key: String): String</code> <code>get(key: String, default: String): String</code>
<code>getAll</code>	<code>getAll: Map[String, String]</code>
<code>getOption</code>	<code>getOption(key: String): Option[String]</code>
<code>isModifiable</code>	<code>isModifiable(key: String): Boolean</code> (New in 2.4.0)
<code>set</code>	<code>set(key: String, value: Boolean): Unit</code> <code>set(key: String, value: Long): Unit</code> <code>set(key: String, value: String): Unit</code>
<code>unset</code>	<code>unset(key: String): Unit</code>

`RuntimeConfig` is available using the `conf` attribute of a `SparkSession`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.conf
org.apache.spark.sql.RuntimeConfig
```



Figure 1. RuntimeConfig, SparkSession and SQLConf

`RuntimeConfig` is [created](#) exclusively when `sparkSession` is requested for `one`.

`RuntimeConfig` takes a [SQLConf](#) when created.

get Method

```
get(key: String): String  
get(key: String, default: String): String
```

`get` ...FIXME

Note	<code>get</code> is used when...FIXME
------	---------------------------------------

getAll Method

```
getAll: Map[String, String]
```

`getAll` ...FIXME

Note	<code>getAll</code> is used when...FIXME
------	--

getOption Method

```
getOption(key: String): Option[String]
```

`getOption` ...FIXME

Note	<code>getOption</code> is used when...FIXME
------	---

set Method

```
set(key: String, value: Boolean): Unit  
set(key: String, value: Long): Unit  
set(key: String, value: String): Unit
```

`set` ...FIXME

Note	<code>set</code> is used when...FIXME
------	---------------------------------------

unset Method

```
unset(key: String): Unit
```

`unset` ...FIXME

Note

`unset` is used when...FIXME

SQLConf—Internal Configuration Store

`SQLConf` is an **internal key-value configuration store** for [parameters and hints](#) used in Spark SQL.

`SQLConf` is an internal part of Spark SQL and is not supposed to be used directly.

Spark SQL configuration is available through [RuntimeConfig](#) (the user-facing configuration management interface) that you can access using [SparkSession](#).

Note

```
scala> :type spark
org.apache.spark.sql.SparkSession
```

```
scala> :type spark.conf
org.apache.spark.sql.RuntimeConfig
```

You can access a `SQLConf` using:

1. [SQLConf.get](#) (preferred) - the `SQLConf` of the current active `SparkSession`
2. [SessionState](#) - direct access through `SessionState` of the `sparkSession` of your choice (that gives more flexibility on what `SparkSession` is used that can be different from the current active `sparkSession`)

```
import org.apache.spark.sql.internal.SQLConf

// Use type-safe access to configuration properties
// using SQLConf.get.getConf
val parallelFileListingInStatsComputation = SQLConf.get.getConf(SQLConf.PARALLEL_FILE_
LISTING_IN_STATS_COMPUTATION)

// or even simpler
SQLConf.get.parallelFileListingInStatsComputation
```

`SQLConf` offers methods to [get](#), [set](#), [unset](#) or [clear](#) values of configuration properties, but has also the [accessor methods](#) to read the current value of a configuration property or hint.

```

scala> :type spark
org.apache.spark.sql.SparkSession

// Direct access to the session SQLConf
val sqlConf = spark.sessionState.conf
scala> :type sqlConf
org.apache.spark.sql.internal.SQLConf

scala> println(sqlConf.offHeapColumnVectorEnabled)
false

// Or simply import the conf value
import spark.sessionState.conf

// accessing properties through accessor methods
scala> conf.numShufflePartitions
res1: Int = 200

// Prefer SQLConf.get (over direct access)
import org.apache.spark.sql.internal.SQLConf
val cc = SQLConf.get
scala> cc == conf
res4: Boolean = true

// setting properties using aliases
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
conf.setConf(SHUFFLE_PARTITIONS, 2)
scala> conf.numShufflePartitions
res2: Int = 2

// unset aka reset properties to the default value
conf.unsetConf(SHUFFLE_PARTITIONS)
scala> conf.numShufflePartitions
res3: Int = 200

```

Table 1. SQLConf's Accessor

Name	Parameter
adaptiveExecutionEnabled	spark.sql.adaptive.enabled
autoBroadcastJoinThreshold	spark.sql.autoBroadcastJoinThreshold
autoSizeUpdateEnabled	spark.sql.statistics.size.autoUpdate.enabled

avroCompressionCodec	spark.sql.avro.compression.codec
broadcastTimeout	spark.sql.broadcastTimeout
bucketingEnabled	spark.sql.sources.bucketing.enabled
cacheVectorizedReaderEnabled	spark.sql.inMemoryColumnarStorage.enableVe
caseSensitiveAnalysis	spark.sql.caseSensitive
cboEnabled	spark.sql.cbo.enabled
columnBatchSize	spark.sql.inMemoryColumnarStorage.batchSize
dataFramePivotMaxValues	spark.sql.pivotMaxValues
dataFrameRetainGroupColumns	spark.sql.retainGroupColumns

defaultSizeInBytes	spark.sql.defaultSizeInBytes
enableRadixSort	spark.sql.sort.enableRadixSort
exchangeReuseEnabled	spark.sql.exchange.reuse
fallBackToHdfsForStatsEnabled	spark.sql.statistics.fallBackToHdfs
fileCommitProtocolClass	spark.sql.sources.commitProtocolClass
filesMaxPartitionBytes	spark.sql.files.maxPartitionBytes
filesOpenCostInBytes	spark.sql.files.openCostInBytes
histogramEnabled	spark.sql.statistics.histogram.enabled

histogramNumBins	spark.sql.statistics.histogram.numBins
hugeMethodLimit	spark.sql_codegen.hugeMethodLimit
ignoreCorruptFiles	spark.sql.files.ignoreCorruptFiles
ignoreMissingFiles	spark.sql.files.ignoreMissingFiles
inMemoryPartitionPruning	spark.sql.inMemoryColumnarStorage.partitionF
isParquetBinaryAsString	spark.sql.parquet.binaryAsString
isParquetINT96AsTimestamp	spark.sql.parquet.int96AsTimestamp
isParquetINT96TimestampConversion	spark.sql.parquet.int96TimestampConversion
joinReorderEnabled	spark.sql.cbo.joinReorder.enabled
limitScaleUpFactor	spark.sql.limit.scaleUpFactor

manageFilesourcePartitions	spark.sql.hive.manageFilesourcePartitions
minNumPostShufflePartitions	spark.sql.adaptive.minNumPostShufflePartition
numShufflePartitions	spark.sql.shuffle.partitions
offHeapColumnVectorEnabled	spark.sql.columnVector.offheap.enabled
optimizerExcludedRules	spark.sql.optimizer.excludedRules
optimizerInSetConversionThreshold	spark.sql.optimizer.inSetConversionThreshold

parallelFileListingInStatsComputation	spark.sql.statistics.parallelFileListingInStatsCor
parquetFilterPushDown	spark.sql.parquet.filterPushdown
parquetFilterPushDownDate	spark.sql.parquet.filterPushdown.date
parquetRecordFilterEnabled	spark.sql.parquet.recordLevelFilter.enabled
parquetVectorizedReaderBatchSize	spark.sql.parquet.columnarReaderBatchSize
parquetVectorizedReaderEnabled	spark.sql.parquet.enableVectorizedReader
partitionOverwriteMode	spark.sql.sources.partitionOverwriteMode
preferSortMergeJoin	spark.sql.join.preferSortMergeJoin
runSQLOnFile	spark.sql.runSQLOnFiles
sessionLocalTimeZone	spark.sql.session.timeZone

starSchemaDetection	spark.sql.cbo.starSchemaDetection
stringRedactionPattern	spark.sql.redaction.string.regex
subexpressionEliminationEnabled	spark.sql.subexpressionElimination.enabled
supportQuotedRegexColumnName	spark.sql.parser.quotedRegexColumnNames
targetPostShuffleInputSize	spark.sql.adaptive.shuffle.targetPostShuffleInputSize
useCompression	spark.sql.inMemoryColumnarStorage.compress
wholeStageEnabled	spark.sql_codegen.wholeStage
wholeStageFallback	spark.sql_codegen.fallback
wholeStageMaxNumFields	spark.sql_codegen.maxFields

<code>wholeStageSplitConsumeFuncByOperator</code>	<code>spark.sqlcodegen.splitConsumeFuncByOperat</code>
<code>wholeStageUseIdInClassName</code>	<code>spark.sqlcodegen.useldInClassName</code>
<code>windowExecBufferInMemoryThreshold</code>	<code>spark.sql.windowExec.buffer.in.memory.thresho</code>
<code>windowExecBufferSpillThreshold</code>	<code>spark.sql.windowExec.buffer.spill.threshold</code>
<code>useObjectHashAggregation</code>	<code>spark.sql.execution.useObjectHashAggregateE</code>

Getting Parameters and Hints

You can get the current parameters and hints using the following family of `get` methods.

```
getConf[T](entry: ConfigEntry[T], defaultValue: T): T
getConf[T](entry: ConfigEntry[T]): T
getConf[T](entry: OptionalConfigEntry[T]): Option[T]
getConfString(key: String): String
getConfString(key: String, defaultValue: String): String
getAllConfs: immutable.Map[String, String]
getAllDefinedConfs: Seq[(String, String, String)]
```

Setting Parameters and Hints

You can set parameters and hints using the following family of `set` methods.

```
setConf(props: Properties): Unit
setConfString(key: String, value: String): Unit
setConf[T](entry: ConfigEntry[T], value: T): Unit
```

Unsetting Parameters and Hints

You can unset parameters and hints using the following family of `unset` methods.

```
unsetConf(key: String): Unit  
unsetConf(entry: ConfigEntry[_]): Unit
```

Clearing All Parameters and Hints

```
clear(): Unit
```

You can use `clear` to remove all the parameters and hints in `SQLConf`.

Redacting Data Source Options with Sensitive Information — `redactOptions` Method

```
redactOptions(options: Map[String, String]): Map[String, String]
```

`redactOptions` takes the values of the `spark.sql.redaction.options.regex` and `spark.redaction.regex` configuration properties.

For every regular expression (in the order), `redactOptions` redacts sensitive information, i.e. finds the first match of a regular expression pattern in every option key or value and if either matches replaces the value with `***(redacted)`.

Note	<code>redactOptions</code> is used exclusively when <code>SaveIntoDataSourceCommand</code> logical command is requested for the simple description .
------	--

`redactOptions` is used exclusively when `SaveIntoDataSourceCommand` logical command is requested for the [simple description](#).

StaticSQLConf — Cross-Session, Immutable and Static SQL Configuration

`StaticSQLConf` holds cross-session, immutable and static SQL configuration properties.

Table 1. StaticSQLConf's Configuration Properties

Name	Default Value	Scala Value
<code>spark.sql.catalogImplementation</code>	in-memory	CATALOG_IMPLEMENTATION
<code>spark.sql.debug</code>	false	DEBUG_MODE
<code>spark.sql.extensions</code>	(empty)	SPARK_SESSION_EXTENSIONS

spark.sql.filesourceTableRelationCacheSize	1000	FILESOURCE_TABLE_RELATION_CACHE_SIZE
spark.sql.globalTempDatabase	global_temp	GLOBAL_TEMP_DATABASE
spark.sql.hive.thriftServer.singleSession	false	HIVE_THRIFT_SERVER_SINGLE_SESSION
spark.sql.queryExecutionListeners	(empty)	QUERY_EXECUTION_LISTENERS
spark.sql.sources.schemaStringLengthThreshold	4000	SCHEMA_STRING_LENGTH_THRESHOLD
spark.sql.ui.retainedExecutions	1000	UI_RETAINED_EXECUTIONS
spark.sql.warehouse.dir	spark-warehouse	WAREHOUSE_PATH

--	--	--

The [properties](#) in `staticSQLConf` can only be queried and can never be changed once the first `SparkSession` is created.

```
import org.apache.spark.sql.internal.StaticSQLConf
scala> val metastoreName = spark.conf.get(StaticSQLConf.CATALOG_IMPLEMENTATION.key)
metastoreName: String = hive

scala> spark.conf.set(StaticSQLConf.CATALOG_IMPLEMENTATION.key, "hive")
org.apache.spark.sql.AnalysisException: Cannot modify the value of a static config: spark.sql.catalogImplementation;
    at org.apache.spark.sql.RuntimeConfig.requireNonStaticConf(RuntimeConfig.scala:144)
    at org.apache.spark.sql.RuntimeConfig.set(RuntimeConfig.scala:41)
    ... 50 elided
```

CatalystConf

`CatalystConf` is...FIXME

Note

The default `CatalystConf` is [SQLConf](#) that is...FIXME

Table 1. CatalystConf's Internal Properties

Name	Initial Value	Description
<code>caseSensitiveAnalysis</code>		
<code>cboEnabled</code>		<p>Enables cost-based optimizations (CBO) for estimation of plan statistics when enabled.</p> <p>Used in CostBasedJoinReorder logical plan optimization and Project , Filter , Join and Aggregate logical operators.</p>
<code>optimizerMaxIterations</code>	<code>spark.sql.optimizer.maxIterations</code>	Maximum number of iterations for Analyzer and Optimizer .
<code>sessionLocalTimeZone</code>		

resolver Method

`resolver` gives case-sensitive or case-insensitive `Resolvers` per `caseSensitiveAnalysis` setting.

Note

`Resolver` is a mere function of two `String` parameters that returns `true` if both refer to the same entity (i.e. for case insensitive equality).

UDFRegistration — Session-Spaced FunctionRegistry

`UDFRegistration` is an interface to the session-scoped [FunctionRegistry](#) to register user-defined functions (UDFs) and [user-defined aggregate functions](#) (UDAFs).

`UDFRegistration` is available using [SparkSession](#).

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
spark.udf
```

`UDFRegistration` takes a [FunctionRegistry](#) when created.

`UDFRegistration` is [created](#) exclusively for [SessionState](#).

Registering UserDefinedFunction (with FunctionRegistry) — register Method

```
register(name: String, func: Function0[RT]): UserDefinedFunction
register(name: String, func: Function1[A1, RT]): UserDefinedFunction
...
register(name: String, func: Function22[A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11,
A12, A13, A14, A15, A16, A17, A18, A19, A20, A21, A22, RT]): UserDefinedFunction
```

`register ...FIXME`

Note

`register` is used when...FIXME

Registering UserDefinedFunction (with FunctionRegistry) — register Method

```
register(name: String, udf: UserDefinedFunction): UserDefinedFunction
```

`register ...FIXME`

Note

`register` is used when...FIXME

Registering UserDefinedAggregateFunction (with FunctionRegistry) — register Method

```
register(  
    name: String,  
    udaf: UserDefinedAggregateFunction): UserDefinedAggregateFunction
```

`register` registers a `UserDefinedAggregateFunction` under `name` with `FunctionRegistry`.

`register` creates a `ScalaUDAF` internally to register a UDAF.

Note	<code>register</code> gives the input <code>udaf</code> aggregate function back after the function has been registered with <code>FunctionRegistry</code> .
------	---

FileFormat — Data Sources to Read and Write Data In Files

`FileFormat` is the [contract](#) for [data sources](#) that [read](#) and [write](#) data stored in files.

Table 1. FileFormat Contract

Method	Description
<code>buildReader</code>	<pre>buildReader(sparkSession: SparkSession, dataSchema: StructType, partitionSchema: StructType, requiredSchema: StructType, filters: Seq[Filter], options: Map[String, String], hadoopConf: Configuration): PartitionedFile => Iterator[InternalRow]</pre> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>buildReader</code> throws an <code>UnsupportedOperationException</code> default (and should therefore be overriden to work): <code>buildReader</code> is not supported for [this] </div> <p>Used exclusively when <code>FileFormat</code> is requested to buildReaderWithPartitionValues</p>
<code>buildReaderWithPartitionValues</code>	<pre>buildReaderWithPartitionValues(sparkSession: SparkSession, dataSchema: StructType, partitionSchema: StructType, requiredSchema: StructType, filters: Seq[Filter], options: Map[String, String], hadoopConf: Configuration): PartitionedFile => Iterator[InternalRow]</pre> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>buildReaderWithPartitionValues</code> builds a data reader where the partition column values are appended, i.e. a function that is used to read a single file in (as a <code>PartitionedFile</code>) as an <code>Iterator</code> of <code>InternalRows</code> (like <code>buildReader</code>) with the partition value appended. </div> <p>Used exclusively when <code>FileSourceScanExec</code> physical operator is requested for the <code>inputRDD</code> (when requesting the <code>inputRDDs</code> and <code>execution</code>)</p>

```
inferSchema(
    sparkSession: SparkSession,
    options: Map[String, String],
    files: Seq[FileStatus]): Option[StructType]
```

Infers (returns) the [schema](#) of the given files (as [HadoopFileStatuses](#)) if supported. Otherwise, `None` should be returned.

`inferSchema`

Used when:

- `HiveMetastoreCatalog` is requested to [inferIfNecessary](#) (when `RelationConversions` logical evaluation rule requested to convert a [HiveTableRelation](#) to a [LogicalRelation](#) for `parquet`, `native` and `hive` storage formats)
- `DataSource` is requested to [getOrInferFileFormatSchema](#) and [resolveRelation](#)

```
isSplitable(
    sparkSession: SparkSession,
    options: Map[String, String],
    path: Path): Boolean
```

`isSplitable`

Controls whether the format (under the given path as [Path](#)) can be split or not.

`isSplitable` is disabled (`false`) by default.

Used exclusively when `FileSourceScanExec` physical operator is requested to [create an RDD for non-bucketed reads](#) (when requested for the `inputRDD` and neither the optional [bucketing specification](#) of the `HadoopFsRelation` defined nor [bucketing is enabled](#))

```
prepareWrite(
    sparkSession: SparkSession,
    job: Job,
    options: Map[String, String],
    dataSchema: StructType): OutputWriterFactory
```

`prepareWrite`

Prepares a write job and returns an `OutputWriterFactory`

Used exclusively when `FileFormatWriter` is requested to [write query result](#)

```
supportBatch(
    sparkSession: SparkSession,
    dataSchema: StructType): Boolean
```

<pre>supportBatch</pre>	<p>Flag that says whether the format supports vectorized decoding (aka <i>columnar batch</i>) or not.</p> <p>Default: <code>false</code></p> <p>Used exclusively when <code>FileSourceScanExec</code> physical operator is requested for the supportsBatch</p>
<pre>vectorTypes()</pre>	<pre>vectorTypes(requiredSchema: StructType, partitionSchema: StructType, sqlConf: SQLConf): Option[Seq[String]]</pre> <p>Defines the fully-qualified class names (<i>types</i>) of the concrete ColumnVectors for every column in the input <code>requiredSchema</code> and <code>partitionSchema</code> schemas that are used in a columnar batch.</p> <p>Default: <code>undefined</code> (<code>None</code>)</p> <p>Used exclusively when <code>FileSourceScanExec</code> leaf physical operator is requested for the vectorTypes</p>

Table 2. FileFormats (Direct Implementations and Extensions)

FileFormat	Description
AvroFileFormat	Avro data source
HiveFileFormat	Writes hive tables
OrcFileFormat	ORC data source
ParquetFileFormat	Parquet data source
TextBasedFileFormat	Base for text splitable <code>FileFormats</code>

Building Data Reader With Partition Column Values Appended — `buildReaderWithPartitionValues` Method

```
buildReaderWithPartitionValues(  
    sparkSession: SparkSession,  
    dataSchema: StructType,  
    partitionSchema: StructType,  
    requiredSchema: StructType,  
    filters: Seq[Filter],  
    options: Map[String, String],  
    hadoopConf: Configuration): PartitionedFile => Iterator[InternalRow]
```

`buildReaderWithPartitionValues` is simply an enhanced `buildReader` that appends **partition column values** to the internal rows produced by the reader function from `buildReader`.

Internally, `buildReaderWithPartitionValues` builds a data reader with the input parameters and gives a **data reader function** (of a `PartitionedFile` to an `Iterator[InternalRow]`) that does the following:

1. Creates a converter by requesting `GenerateUnsafeProjection` to generate an `UnsafeProjection` for the attributes of the input `requiredSchema` and `partitionSchema`
2. Applies the data reader to a `PartitionedFile` and converts the result using the converter on the joined row with the **partition column values** appended.

Note

`buildReaderWithPartitionValues` is used exclusively when `FileSourceScanExec` physical operator is requested for the `input RDDs`.

OrcFileFormat

OrcFileFormat is a [FileFormat](#) that...FIXME

buildReaderWithPartitionValues Method

```
buildReaderWithPartitionValues(  
    sparkSession: SparkSession,  
    dataSchema: StructType,  
    partitionSchema: StructType,  
    requiredSchema: StructType,  
    filters: Seq[Filter],  
    options: Map[String, String],  
    hadoopConf: Configuration): (PartitionedFile) => Iterator[InternalRow]
```

Note

`buildReaderWithPartitionValues` is part of [FileFormat Contract](#) to build a data reader with partition column values appended.

`buildReaderWithPartitionValues` ...FIXME

inferSchema Method

```
inferSchema(  
    sparkSession: SparkSession,  
    options: Map[String, String],  
    files: Seq[FileStatus]): Option[StructType]
```

Note

`inferSchema` is part of [FileFormat Contract](#) to...FIXME.

`inferSchema` ...FIXME

Building Partitioned Data Reader — `buildReader` Method

```
buildReader(  
    sparkSession: SparkSession,  
    dataSchema: StructType,  
    partitionSchema: StructType,  
    requiredSchema: StructType,  
    filters: Seq[Filter],  
    options: Map[String, String],  
    hadoopConf: Configuration): (PartitionedFile) => Iterator[InternalRow]
```

Note

`buildReader` is part of [FileFormat Contract](#) to...FIXME

`buildReader` ...FIXME

ParquetFileFormat

`ParquetFileFormat` is the [FileFormat](#) for **parquet** data source (i.e. registers itself to handle files in parquet format and converts them to Spark SQL rows).

Note

`parquet` is the [default data source format](#) in Spark SQL.

Note

[Apache Parquet](#) is a columnar storage format for the Apache Hadoop ecosystem with support for efficient storage and encoding of data.

```
// All the following queries are equivalent
// schema has to be specified manually
import org.apache.spark.sql.types.StructType
val schema = StructType($"id".int :: Nil)

spark.read.schema(schema).format("parquet").load("parquet-datasets")

// The above is equivalent to the following shortcut
// Implicitly does format("parquet").load
spark.read.schema(schema).parquet("parquet-datasets")

// parquet is the default data source format
spark.read.schema(schema).load("parquet-datasets")
```

`ParquetFileFormat` is [splittable](#), i.e. `FIXME`

`ParquetFileFormat` supports vectorized parquet decoding in whole-stage code generation when all of the following hold:

1. `spark.sql.parquet.enableVectorizedReader` configuration property is enabled
2. `spark.sqlcodegen.wholeStage` internal configuration property is enabled
3. The number of fields in the schema is at most `spark.sqlcodegen.maxFields` internal configuration property
4. All the fields in the output schema are of [AtomicType](#)

`ParquetFileFormat` supports **filter predicate push-down optimization** (via `createFilter`) as per the following [table](#).

Table 1. Spark Data Source Filters to Parquet Filter Predicates Conversions (aka
`ParquetFilters.createFilter()`)

Data Source Filter	Parquet FilterPredicate
<code>IsNull</code>	<code>FilterApi.eq</code>
<code>IsNotNull</code>	<code>FilterApi.notEq</code>
<code>EqualTo</code>	<code>FilterApi.eq</code>
<code>Not EqualTo</code>	<code>FilterApi.notEq</code>
<code>EqualNullSafe</code>	<code>FilterApi.eq</code>
<code>Not EqualNullSafe</code>	<code>FilterApi.notEq</code>
<code>LessThan</code>	<code>FilterApi.lt</code>
<code>LessThanOrEqual</code>	<code>FilterApi.ltEq</code>
<code>Greater Than</code>	<code>FilterApi.gt</code>
<code>Greater ThanOrEqual</code>	<code>FilterApi.gtEq</code>
<code>And</code>	<code>FilterApi.and</code>
<code>Or</code>	<code>FilterApi.or</code>
<code>No</code>	<code>FilterApi.not</code>

Tip	<p>Enable <code>ALL</code> logging level for <code>org.apache.spark.sql.execution.datasources.parquet.ParquetFileFormat</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.datasources.parquet.ParquetFileFormat=ALL</pre> <p>Refer to Logging.</p>
-----	---

Preparing Write Job— `prepareWrite` Method

```
prepareWrite(
    sparkSession: SparkSession,
    job: Job,
    options: Map[String, String],
    dataSchema: StructType): OutputWriterFactory
```

Note	<code>preparewrite</code> is part of the FileFormat Contract to prepare a write job.
------	--

`prepareWrite ...FIXME`

inferSchema Method

```
inferSchema(  
    sparkSession: SparkSession,  
    parameters: Map[String, String],  
    files: Seq[FileStatus]): Option[StructType]
```

Note	<code>inferSchema</code> is part of FileFormat Contract to...FIXME.
------	---

`inferSchema ...FIXME`

vectorTypes Method

```
vectorTypes(  
    requiredSchema: StructType,  
    partitionSchema: StructType,  
    sqlConf: SQLConf): Option[Seq[String]]
```

Note	<code>vectorTypes</code> is part of FileFormat Contract to define the concrete column vector class names for each column used in a columnar batch when enabled .
------	--

`vectorTypes` creates a collection of the names of [OffHeapColumnVector](#) or [OnHeapColumnVector](#) when `spark.sql.columnVector.offheap.enabled` property is enabled or disabled, respectively.

Note	<code>spark.sql.columnVector.offheap.enabled</code> property is disabled (<code>false</code>) by default.
------	---

The size of the collection are all the fields of the given `requiredSchema` and `partitionSchema` schemas.

Building Data Reader With Partition Column Values Appended — `buildReaderWithPartitionValues` Method

```
buildReaderWithPartitionValues(
    sparkSession: SparkSession,
    dataSchema: StructType,
    partitionSchema: StructType,
    requiredSchema: StructType,
    filters: Seq[Filter],
    options: Map[String, String],
    hadoopConf: Configuration): (PartitionedFile) => Iterator[InternalRow]
```

Note `buildReaderWithPartitionValues` is part of [FileFormat Contract](#) to build a data reader with the partition column values appended.

`buildReaderWithPartitionValues` sets the [configuration options](#) in the input `hadoopConf`.

Table 2. Hadoop Configuration Options

Name	Value
<code>parquet.read.support.class</code>	ParquetReadSupport
<code>org.apache.spark.sql.parquet.row.requested_schema</code>	JSON representation of requiredSchema
<code>org.apache.spark.sql.parquet.row.attributes</code>	JSON representation of requiredSchema
<code>spark.sql.session.timeZone</code>	spark.sql.session.timeZone
<code>spark.sql.parquet.binaryAsString</code>	spark.sql.parquet.binaryAsString
<code>spark.sql.parquet.int96AsTimestamp</code>	spark.sql.parquet.int96AsTimestamp

`buildReaderWithPartitionValues` requests `ParquetWriteSupport` to `setSchema`.

`buildReaderWithPartitionValues` tries to push filters down to create a Parquet `FilterPredicate` (aka pushed).

Note Filter predicate push-down optimization for parquet data sources uses `spark.sql.parquet.filterPushdown` configuration property (default: enabled).

With `spark.sql.parquet.filterPushdown` configuration property enabled, `buildReaderWithPartitionValues` takes the input Spark data source `filters` and converts them to Parquet filter predicates if possible (as described in the [table](#)). Otherwise, the Parquet filter predicate is not specified.

Note `buildReaderWithPartitionValues` creates filter predicates for the following types: `BooleanType`, `IntegerType`, `LongType`, `FloatType`, `DoubleType`, `StringType`, `BinaryType`.

`buildReaderWithPartitionValues` broadcasts the input `hadoopConf` Hadoop Configuration .

In the end, `buildReaderWithPartitionValues` gives a function that takes a [PartitionedFile](#) and does the following:

1. Creates a Hadoop `FileSplit` for the input `PartitionedFile`
2. Creates a Parquet `ParquetInputSplit` for the Hadoop `FileSplit` created
3. Gets the broadcast Hadoop `Configuration`
4. Creates a flag that says whether to apply timezone conversions to int96 timestamps or not (aka `convertTz`)
5. Creates a Hadoop `TaskAttemptContextImpl` (with the broadcast Hadoop `Configuration` and a Hadoop `TaskAttemptID` for a map task)
6. Sets the Parquet `FilterPredicate` (only when `spark.sql.parquet.filterPushdown` configuration property is enabled and it is by default)

The function then branches off on whether [Parquet vectorized reader](#) is enabled or not.

Note	Parquet vectorized reader is enabled by default.
------	--

With [Parquet vectorized reader](#) enabled, the function does the following:

1. Creates a [VectorizedParquetRecordReader](#) and a [RecordReaderIterator](#)
2. Requests `VectorizedParquetRecordReader` to [initialize](#) (with the Parquet `ParquetInputSplit` and the Hadoop `TaskAttemptContextImpl`)
3. Prints out the following DEBUG message to the logs:

Appending [partitionSchema] [partitionValues]
4. Requests `VectorizedParquetRecordReader` to [initBatch](#)
5. (only with `supportBatch` enabled) Requests `VectorizedParquetRecordReader` to [enableReturningBatches](#)
6. In the end, the function gives the [RecordReaderIterator](#) (over the `VectorizedParquetRecordReader`) as the `Iterator[InternalRow]`

With [Parquet vectorized reader](#) disabled, the function does the following:

1. `FIXME` (since Parquet vectorized reader is enabled by default it's of less interest currently)

mergeSchemasInParallel Method

```
mergeSchemasInParallel(  
    filesToTouch: Seq[FileStatus],  
    sparkSession: SparkSession): Option[StructType]
```

mergeSchemasInParallel ...FIXME

Note

mergeSchemasInParallel is used when...FIXME

TextBasedFileFormat — Base for Text Splittable FileFormats

`TextBasedFileFormat` is an extension of the [FileFormat](#) contract for [formats](#) that can be [splittable](#).

Table 1. TextBasedFileFormats

TextBasedFileFormat	Description
CSVFileFormat	
JsonFileFormat	
LibSVMFileFormat	Used in Spark MLlib
TextFileFormat	

`TextBasedFileFormat` uses Hadoop's [CompressionCodecFactory](#) to find the proper compression codec for the given file.

isSplittable Method

```
isSplittable(
    sparkSession: SparkSession,
    options: Map[String, String],
    path: Path): Boolean
```

Note	<code>isSplittable</code> is part of FileFormat Contract to know whether a given file is splittable or not.
------	---

`isSplittable` requests the [CompressionCodecFactory](#) to find the [compression codec](#) for the given file (as the input `path`) based on its filename suffix.

`isSplittable` returns `true` when the compression codec is not used (i.e. `null`) or is a Hadoop [SplittableCompressionCodec](#) (e.g. [BZip2Codec](#)).

If the [CompressionCodecFactory](#) is not defined, `isSplittable` creates a [CompressionCodecFactory](#) (with a Hadoop `Configuration` by requesting the `SessionState` for a [new Hadoop Configuration with extra options](#)).

Note	<code>isSplittable</code> uses the input <code>sparkSession</code> to access SessionState .
------	---

	<p>SplittableCompressionCodec interface is for compression codecs that are capable to compress and de-compress a stream starting at any arbitrary position.</p> <p>Note Such codecs are highly valuable, especially in the context of Hadoop, because an input compressed file can be split and hence can be worked on by multiple machines in parallel.</p> <p>One such compression codec is BZip2Codec that provides output and input streams for bzip2 compression and decompression.</p>
--	--

CSVFileFormat

`CSVFileFormat` is a [TextBasedFileFormat](#) for `csv` format (i.e. registers itself to handle files in `csv` format and converts them to Spark SQL rows).

```
spark.read.format("csv").load("csv-datasets")

// or the same as above using a shortcut
spark.read.csv("csv-datasets")
```

`CSVFileFormat` uses [CSV options](#) (that in turn are used to configure the underlying CSV parser from [uniVocity-parsers](#) project).

Table 1. CSVFileFormat's Options

Option	Default Value	Description
<code>charset</code>	<code>UTF-8</code>	Alias of encoding
<code>charToEscapeQuoteEscaping</code>	<code>\\"</code>	One character to...FIXME
<code>codec</code>		Compression codec that can be either one of the known aliases or a fully-qualified class name. Alias of compression
<code>columnNameOfCorruptRecord</code>		
<code>comment</code>	<code>\u0000</code>	
<code>compression</code>		Compression codec that can be either one of the known aliases or a fully-qualified class name. Alias of codec
<code>dateFormat</code>	<code>yyyy-MM-dd</code>	Uses <code>en_US</code> locale
<code>delimiter</code>	<code>,</code> (comma)	Alias of sep
<code>encoding</code>	<code>UTF-8</code>	Alias of charset
<code>escape</code>	<code>\\"</code>	

escapeQuotes	true	
header		
ignoreLeadingWhiteSpace	<ul style="list-style-type: none"> false (for reading) true (for writing) 	
ignoreTrailingWhiteSpace	<ul style="list-style-type: none"> false (for reading) true (for writing) 	
inferSchema		
maxCharsPerColumn	-1	
maxColumns	20480	
mode	PERMISSIVE	<p>Possible values:</p> <ul style="list-style-type: none"> DROPMALFORMED PERMISSIVE (default) FAILFAST
multiLine	false	
nanValue	NaN	
negativeInf	-Inf	
nullValue	(empty string)	
positiveInf	Inf	
sep	, (comma)	Alias of delimiter
timestampFormat	yyyy-MM-dd'T'HH:mm:ss.SSSXXX	Uses timeZone and en_US locale
timeZone	spark.sql.session.timeZone	
quote	\"	
quoteAll	false	

Preparing Write Job— `prepareWrite` Method

```
prepareWrite(  
    sparkSession: SparkSession,  
    job: Job,  
    options: Map[String, String],  
    dataSchema: StructType): OutputWriterFactory
```

Note

`prepareWrite` is part of the [FileFormat Contract](#) to prepare a write job.

`prepareWrite ...FIXME`

Building Partitioned Data Reader— `buildReader` Method

```
buildReader(  
    sparkSession: SparkSession,  
    dataSchema: StructType,  
    partitionSchema: StructType,  
    requiredSchema: StructType,  
    filters: Seq[Filter],  
    options: Map[String, String],  
    hadoopConf: Configuration): (PartitionedFile) => Iterator[InternalRow]
```

Note

`buildReader` is part of the [FileFormat Contract](#) to build a [PartitionedFile](#) reader.

`buildReader ...FIXME`

JsonFileFormat — Built-In Support for Files in JSON Format

`JsonFileFormat` is a [TextBasedFileFormat](#) for **json** format (i.e. registers itself to handle files in [json format](#) and convert them to Spark SQL rows).

```
spark.read.format("json").load("json-datasets")

// or the same as above using a shortcut
spark.read.json("json-datasets")
```

`JsonFileFormat` comes with [options](#) to further customize JSON parsing.

Note

`JsonFileFormat` uses [Jackson 2.6.7](#) as the JSON parser library and some [options](#) map directly to Jackson's internal options (as `JsonParser.Feature`).

Table 1. `JsonFileFormat`'s Options

Option	Default Value	Note	Internally, all options map directly to Jackson's internal options (as <code>JsonParser.Feature</code>)
<code>allowBackslashEscapingAnyCharacter</code>	false		
<code>allowComments</code>	false		
<code>allowNonNumericNumbers</code>	true		
<code>allowNumericLeadingZeros</code>	false		
<code>allowSingleQuotes</code>	true		
<code>allowUnquotedControlChars</code>	false		
<code>allowUnquotedFieldNames</code>	false		

columnNameOfCorruptRecord		
compression		Compression codec that fully-qualified class name
dateFormat	yyyy-MM-dd	Date format Note Internally, <code>dateLang's</code> <code>FastDateParser</code>
multiLine	false	Controls whether...FIXME
mode	PERMISSIVE	Case insensitive name of <ul style="list-style-type: none">• PERMISSIVE• DROPMALFORMED• FAILFAST
prefersDecimal	false	
primitivesAsString	false	
samplingRatio	1.0	
timestampFormat	yyyy-MM-dd'T'HH:mm:ss.SSSXXX	Timestamp format Note Internally, <code>timeZone</code> <code>Commons Lang</code>
timeZone		Java's <code>TimeZone</code>

isSplitable Method

```
isSplitable(  
    sparkSession: SparkSession,  
    options: Map[String, String],  
    path: Path): Boolean
```

Note	<code>isSplitable</code> is part of FileFormat Contract .
------	---

`isSplitable` ...FIXME

inferSchema Method

```
inferSchema(  
    sparkSession: SparkSession,  
    options: Map[String, String],  
    files: Seq[FileStatus]): Option[StructType]
```

Note

`inferSchema` is part of [FileFormat Contract](#).

`inferSchema` ...FIXME

Building Partitioned Data Reader— `buildReader` Method

```
buildReader(  
    sparkSession: SparkSession,  
    dataSchema: StructType,  
    partitionSchema: StructType,  
    requiredSchema: StructType,  
    filters: Seq[Filter],  
    options: Map[String, String],  
    hadoopConf: Configuration): (PartitionedFile) => Iterator[InternalRow]
```

Note

`buildReader` is part of the [FileFormat Contract](#) to build a [PartitionedFile](#) reader.

`buildReader` ...FIXME

Preparing Write Job— `prepareWrite` Method

```
prepareWrite(  
    sparkSession: SparkSession,  
    job: Job,  
    options: Map[String, String],  
    dataSchema: StructType): OutputWriterFactory
```

Note

`prepareWrite` is part of the [FileFormat Contract](#) to prepare a write job.

`prepareWrite` ...FIXME

TextFileFormat

`TextFileFormat` is a [TextBasedFileFormat](#) for **text** format.

```
spark.read.format("text").load("text-datasets")

// or the same as above using a shortcut
spark.read.text("text-datasets")
```

`TextFileFormat` uses [text options](#) while loading a dataset.

Table 1. `TextFileFormat`'s Options

Option	Default Value	Description
<code>compression</code>		Compression codec that can be either one of the known aliases or a fully-qualified class name.
<code>wholetext</code>	<code>false</code>	Enables loading a file as a single row (i.e. not splitting by " <code>\n</code> ")

prepareWrite Method

```
prepareWrite(
    sparkSession: SparkSession,
    job: Job,
    options: Map[String, String],
    dataSchema: StructType): OutputWriterFactory
```

Note

`prepareWrite` is part of [FileFormat Contract](#) that is used when `FileFormatWriter` is requested to write the result of a structured query.

`prepareWrite ...FIXME`

Building Partitioned Data Reader — `buildReader` Method

```
buildReader(  
    sparkSession: SparkSession,  
    dataSchema: StructType,  
    partitionSchema: StructType,  
    requiredSchema: StructType,  
    filters: Seq[Filter],  
    options: Map[String, String],  
    hadoopConf: Configuration): (PartitionedFile) => Iterator[InternalRow]
```

Note

`buildReader` is part of [FileFormat Contract](#) to...FIXME

`buildReader` ...FIXME

readToUnsafeMem Internal Method

```
readToUnsafeMem(  
    conf: Broadcast[SerializableConfiguration],  
    requiredSchema: StructType,  
    wholeTextMode: Boolean): (PartitionedFile) => Iterator[UnsafeRow]
```

`readToUnsafeMem` ...FIXME

Note

`readToUnsafeMem` is used exclusively when `TextFileFormat` is requested to buildReader

JsonDataSource

Caution	FIXME
---------	-------

FileCommitProtocol Contract

`FileCommitProtocol` is the [abstraction](#) of [FIXME](#) that can [FIXME](#).

Table 1. FileCommitProtocol Contract

Method	Description
<code>abortJob</code>	<pre>abortJob(jobContext: JobContext): Unit</pre> <p>Used when...FIXME</p>
<code>abortTask</code>	<pre>abortTask(taskContext: TaskAttemptContext): Unit</pre> <p>Used when...FIXME</p>
<code>commitJob</code>	<pre>commitJob(jobContext: JobContext, taskCommits: Seq[TaskCommitMessage]): Unit</pre> <p>Used when...FIXME</p>
<code>commitTask</code>	<pre>commitTask(taskContext: TaskAttemptContext): TaskCommitMessage</pre> <p>Used when...FIXME</p>
<code>newTaskTempFile</code>	<pre>newTaskTempFile(taskContext: TaskAttemptContext, dir: Option[String], ext: String): String</pre> <p>Used when...FIXME</p>
<code>newTaskTempFileAbsPath</code>	<pre>newTaskTempFileAbsPath(taskContext: TaskAttemptContext, absoluteDir: String, ext: String): String</pre> <p>Used when...FIXME</p>
<code>onTaskCommit</code>	<pre>onTaskCommit(taskCommit: TaskCommitMessage): Unit = {}</pre> <p>Used when...FIXME</p>

setupJob	setupJob(jobContext: JobContext): Unit Used when...FIXME
setupTask	setupTask(taskContext: TaskAttemptContext): Unit Used when...FIXME
setupJob	setupJob(jobContext: JobContext): Unit Used when...FIXME

Table 2. FileCommitProtocols (Direct Implementations and Extensions)

FileCommitProtocol	Description
HadoopMapReduceCommitProtocol	
ManifestFileCommitProtocol	

Tip	Enable ALL logging level for <code>org.apache.spark.internal.io.FileCommitProtocol</code> logger to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.internal.io.FileCommitProtocol=ALL</code>
	Refer to Logging .

Creating FileCommitProtocol Instance (Given Class Name) — `instantiate` Object Method

```
instantiate(  
  className: String,  
  jobId: String,  
  outputPath: String,  
  dynamicPartitionOverwrite: Boolean = false): FileCommitProtocol
```

`instantiate` prints out the following DEBUG message to the logs:

```
Creating committer [className]; job [jobId]; output=[outputPath]; dynamic=[dynamicPart  
itionOverwrite]
```

`instantiate` creates an instance of `FileCommitProtocol` for the given fully-qualified `className` using either 3-argument or 2-argument constructor and prints out the following DEBUG messages to the logs per the argument variant:

```
Using (String, String, Boolean) constructor
```

```
Falling back to (String, String) constructor
```

Note	<p><code>instantiate</code> is used when:</p> <ul style="list-style-type: none">• <code>InsertIntoHadoopFsRelationCommand</code> logical command is executed• <code>SaveAsHiveFile</code> is requested to saveAsHiveFile• <code>HadoopMapRedWriteConfigUtil</code> and <code>HadoopMapReduceWriteConfigUtil</code> are requested to <code>createCommitter</code>• Spark Structured Streaming's <code>FileStreamSink</code> is requested to <code>addBatch</code>
------	---

SQLHadoopMapReduceCommitProtocol

SQLHadoopMapReduceCommitProtocol is...FIXME

PartitionedFile — File Block in FileFormat Data Source

`PartitionedFile` is a part (*block*) of a file that is in a sense similar to a Pqruet block or a HDFS split.

`PartitionedFile` represents a chunk of a file that will be read, along with [partition column values](#) appended to each row, in a partition.

Note

Partition column values are values of the columns that are column partitions and therefore part of the directory structure not the partitioned files themselves (that together are the partitioned dataset).

`PartitionedFile` is [created](#) exclusively when `FileSourceScanExec` is requested to create the input RDD for [bucketed](#) or [non-bucketed](#) reads.

`PartitionedFile` takes the following to be created:

- Partition column values to be appended to each row (as an [internal row](#))
- Path of the file to read
- Beginning offset (in bytes)
- Number of bytes to read (aka `length`)
- Locality information that is a list of nodes (by their host names) that have the data (`Array[String]`). Default: empty

```
import org.apache.spark.sql.execution.datasources.PartitionedFile
import org.apache.spark.sql.catalyst.InternalRow

val partFile = PartitionedFile(InternalRow.empty, "fakePath0", 0, 10, Array("host0", "host1"))
```

`PartitionedFile` uses the following [text representation](#) (`toString`):

```
path: [filePath], range: [start]-[end], partition values: [partitionValues]
```

```
scala> :type partFile
org.apache.spark.sql.execution.datasources.PartitionedFile

scala> println(partFile)
path: fakePath0, range: 0-10, partition values: [empty row]
```

FileScanRDD — Input RDD of FileSourceScanExec Physical Operator

`FileScanRDD` is an `RDD` of [internal binary rows](#) (i.e. `RDD[InternalRow]`) that is the input RDD of a [FileSourceScanExec](#) physical operator (in [Whole-Stage Java Code Generation](#)).

`FileScanRDD` is [created](#) exclusively when `FileSourceScanExec` physical operator is requested to `createBucketedReadRDD` or `createNonBucketedReadRDD` (when `FileSourceScanExec` operator is requested for the `input RDD` when `WholeStageCodegenExec` physical operator is [executed](#)).

```
val q = spark.read.text("README.md")

val sparkPlan = q.queryExecution.executedPlan
import org.apache.spark.sql.execution.FileSourceScanExec
val scan = sparkPlan.collectFirst { case exec: FileSourceScanExec => exec }.get
val inputRDD = scan.inputRDDs.head

import org.apache.spark.sql.execution.datasources.FileScanRDD
assert(inputRDD.isInstanceOf[FileScanRDD])

val rdd = scan.execute
scala> println(rdd.toDebugString)
(1) MapPartitionsRDD[1] at execute at <console>:27 []
 | FileScanRDD[0] at inputRDDs at <console>:26 []

val fileScanRDD = rdd.dependencies.head.rdd
assert(fileScanRDD.isInstanceOf[FileScanRDD])
```

When [created](#), `FileScanRDD` is given [FilePartitions](#) that are custom RDD partitions with [PartitionedFiles](#) (*file blocks*).

`FileScanRDD` uses the following properties when requested to compute a partition:

- `spark.sql.files.ignoreCorruptFiles`
- `spark.sql.files.ignoreMissingFiles`

`FileScanRDD` takes the following to be created:

- `SparkSession`
- Read function that takes a [PartitionedFile](#) and gives [internal rows](#) back
`((PartitionedFile) => Iterator[InternalRow])`
- [FilePartitions](#) (*file blocks*)

Enable `ALL` logging level for `org.apache.spark.sql.execution.datasources.FileScanRDD` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.datasources.FileScanRDD=ALL
```

Refer to [Logging](#).

Placement Preferences of Partition (Preferred Locations) — `getPreferredLocations` Method

```
getPreferredLocations(split: RDDPartition): Seq[String]
```

Note

`getPreferredLocations` is part of the RDD Contract to specify **placement preferences** (aka *preferred locations*) of a partition.

Find out more in [The Internals of Apache Spark](#).

`getPreferredLocations` requests the given `FilePartition` (`split`) for `PartitionedFiles`.

For every `PartitionedFile`, `getPreferredLocations` adds the size of the file(s) to the host (location) it is available at.

In the end, `getPreferredLocations` gives the top 3 hosts with the most data available (file blocks).

RDD Partitions — `getPartitions` Method

```
getPartitions: Array[RDDPartition]
```

Note

`getPartitions` is part of the RDD Contract to specify the partitions of a distributed computation.

Find out more in [The Internals of Apache Spark](#).

`getPartitions` simply returns the `FilePartitions` (the `FileScanRDD` was created with).

Computing Partition (in TaskContext) — `compute` Method

```
compute(split: RDDPartition, context: TaskContext): Iterator[InternalRow]
```

Note

`compute` is part of Spark Core's `RDD` Contract to compute a partition (in a `TaskContext`).

`compute` creates a Scala `Iterator` (of Java `objects`) that...FIXME

Note

The given `RDDPartition` is actually a `FilePartition` with one or more `PartitionedFiles` (*file blocks*).

`compute` then requests the input `TaskContext` to register a completion listener to be executed when a task completes (i.e. `addTaskCompletionListener`) that simply closes the iterator.

In the end, `compute` returns the iterator.

Getting Next Element — `next` Method

```
next(): Object
```

Note

`next` is part of the <<<https://www.scala-lang.org/api/2.12.x/scala/collection/Iterator.html#next>, Iterator Contract>> to produce the next element of this iterator.

`next` takes the next element of the current iterator over elements of a file block (`PartitionedFile`).

`next` increments the metrics of bytes and number of rows read (that could be the number of rows in a `ColumnarBatch` for vectorized reads).

Getting Next Iterator — `nextIterator` Internal Method

```
nextIterator(): Boolean
```

`nextIterator` ...FIXME

Getting Iterator (of Elements) of Current File Block — `readCurrentFile` Internal Method

```
readCurrentFile(): Iterator[InternalRow]
```

```
readCurrentFile ...FIXME
```

ParquetReadSupport — Non-Vectorized ReadSupport in Parquet Data Source

`ParquetReadSupport` is a concrete `ReadSupport` (from Apache Parquet) of [UnsafeRows](#).

`ParquetReadSupport` is [created](#) exclusively when `ParquetFileFormat` is requested for a [data reader](#) (with no support for [Vectorized Parquet Decoding](#) and so falling back to `parquet-mr`).

`ParquetReadSupport` is registered as the fully-qualified class name for `parquet.read.support.class` Hadoop configuration when `ParquetFileFormat` is requested for a [data reader](#).

`ParquetReadSupport` takes an optional Java `TimeZone` to be created.

Tip Enable `ALL` logging level for `org.apache.spark.sql.execution.datasources.parquet.ParquetReadSupport` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.datasources.parquet.ParquetReadSuppo
```

Refer to [Logging](#).

Initializing ReadSupport— `init` Method

```
init(context: InitContext): ReadContext
```

Note	<code>init</code> is part of the <code>ReadSupport</code> Contract to...FIXME.
-------------	--

`init` ...FIXME

prepareForRead Method

```
prepareForRead(
  conf: Configuration,
  keyValueMetaData: JMap[String, String],
  fileSchema: MessageType,
  readContext: ReadContext): RecordMaterializer[UnsafeRow]
```

Note

`prepareForRead` is part of the `ReadSupport` Contract to...FIXME.

`prepareForRead` ...FIXME

RecordReaderIterator — Scala Iterator over Hadoop RecordReader's Values

`RecordReaderIterator` is a Scala `scala.collection.Iterator` over the values of a Hadoop `RecordReader`.

`RecordReaderIterator` is [created](#) when:

- New `OrcFileFormat` and `ParquetFileFormat` are requested to build a data reader
- `HadoopFileLinesReader` and `HadoopFileWholeTextReader` are requested for an value iterator
- Legacy `OrcFileFormat` is requested to [build a data reader](#)

When requested to close, `RecordReaderIterator` simply requests the underlying `RecordReader` to close.

When requested to check whether or not there more internal rows, `RecordReaderIterator` simply requests the underlying `RecordReader` for `nextKeyValue`.

When requested for the next internal row, `RecordReaderIterator` simply requests the underlying `RecordReader` for `getCurrentValue`.

Kafka Data Source

Spark SQL supports [reading](#) data from or [writing](#) data to one or more topics in Apache Kafka.

Note

Apache Kafka is a storage of records in a format-independent and fault-tolerant durable way.

Read up on Apache Kafka in the [official documentation](#) or in my other gitbook [Mastering Apache Kafka](#).

Kafka Data Source supports [options](#) to get better performance of structured queries that use it.

Reading Data from Kafka Topics

As a Spark developer, you use [DataFrameReader.format](#) method to specify Apache Kafka as the external data source to load data from.

You use [kafka](#) (or `org.apache.spark.sql.kafka010.KafkaSourceProvider`) as the input data source format.

```
val kafka = spark.read.format("kafka").load  
  
// Alternatively  
val kafka = spark.read.format("org.apache.spark.sql.kafka010.KafkaSourceProvider").load
```



These one-liners create a [DataFrame](#) that represents the distributed process of loading data from one or many Kafka topics (with additional properties).

Writing Data to Kafka Topics

As a Spark developer,...FIXME

Kafka Data Source Options

Table 1. Kafka Data Source Options

Option	Default	Description
<code>assign</code>		<p>One of the three subscription strategy options (with subscribe and subscribepattern)</p> <p>See KafkaSourceProvider.strategy</p>
<code>endingoffsets</code>		
<code>failondataloss</code>		
<code>kafkaConsumer.pollTimeoutMs</code>		<p>See kafkaConsumer.pollTimeoutMs</p>
<code>startingoffsets</code>		
<code>subscribe</code>		<p>One of the three subscription strategy options (with subscribepattern and assign)</p> <p>See KafkaSourceProvider.strategy</p>
<code>subscribepattern</code>		<p>One of the three subscription strategy options (with subscribe and assign)</p> <p>See KafkaSourceProvider.strategy</p>
<code>topic</code>		<p>Required for writing a DataFrame to Kafka</p> <p>Used when:</p> <ul style="list-style-type: none"> • KafkaSourceProvider is requested to write a DataFrame to a Kafka topic and create a BaseRelation afterwards • (Spark Structured Streaming) KafkaSourceProvider is requested to createStreamWriter and createSink

KafkaSourceProvider

`KafkaSourceProvider` is a [DataSourceRegister](#) and registers itself to handle **kafka** data source format.

Note	<code>KafkaSourceProvider</code> uses META-INF/services/org.apache.spark.sql.sources.DataSourceRegister file for the registration which is available in the source code of Apache Spark.
-------------	--

`KafkaSourceProvider` is a [RelationProvider](#) and a [CreatableRelationProvider](#).

```
// start Spark application like spark-shell with the following package
// --packages org.apache.spark:spark-sql-kafka-0-10_2.12:2.4.4
scala> val fromKafkaTopic1 = spark.
  read.
  format("kafka").
  option("subscribe", "topic1"). // subscribe, subscribepattern, or assign
  option("kafka.bootstrap.servers", "localhost:9092").
  load("gauge_one")
```

`KafkaSourceProvider` uses a fixed schema (and makes sure that a user did not set a custom one).

```
import org.apache.spark.sql.types.StructType
val schema = new StructType().add($"id".int)
scala> spark
  .read
  .format("kafka")
  .option("subscribe", "topic1")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .schema(schema) // <-- defining a custom schema is not supported
  .load
org.apache.spark.sql.AnalysisException: kafka does not allow user-specified schemas.;
  at org.apache.spark.sql.execution.datasources.DataSource.resolveRelation(DataSource.
scala:307)
  at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:178)
  at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:146)
  ... 48 elided
```

Note	<code>KafkaSourceProvider</code> is also a <code>StreamSourceProvider</code> , a <code>StreamSinkProvider</code> , a <code>StreamWriterSupport</code> and a <code>ContinuousReadSupport</code> that are contracts used in Spark Structured Streaming.
-------------	---

You can find more on Spark Structured Streaming in my gitbook [Spark Structured Streaming](#).

Creating BaseRelation — `createRelation` Method (from RelationProvider)

```
createRelation(  
    sqlContext: SQLContext,  
    parameters: Map[String, String]): BaseRelation
```

Note `createRelation` is part of [RelationProvider Contract](#) to create a [BaseRelation](#) (for reading or writing).

`createRelation` starts by [validating the Kafka options \(for batch queries\)](#) in the input `parameters`.

`createRelation` collects all `kafka.`-prefixed key options (in the input `parameters`) and creates a local `specifiedKafkaParams` with the keys without the `kafka.` prefix (e.g. `kafka.whatever` is simply `whatever`).

`createRelation` gets the desired [KafkaOffsetRangeLimit](#) with the `startingoffsets` offset option key (in the given `parameters`) and [EarliestOffsetRangeLimit](#) as the default offsets.

`createRelation` makes sure that the [KafkaOffsetRangeLimit](#) is not [EarliestOffsetRangeLimit](#) or throws an `AssertionError`.

`createRelation` gets the desired [KafkaOffsetRangeLimit](#), but this time with the `endingoffsets` offset option key (in the given `parameters`) and [LatestOffsetRangeLimit](#) as the default offsets.

`createRelation` makes sure that the [KafkaOffsetRangeLimit](#) is not [EarliestOffsetRangeLimit](#) or throws a `AssertionError`.

In the end, `createRelation` creates a [KafkaRelation](#) with the [subscription strategy](#) (in the given `parameters`), [failOnDataLoss](#) option, and the starting and ending offsets.

Validating Kafka Options (for Batch Queries) — `validateBatchOptions` Internal Method

```
validateBatchOptions(caseInsensitiveParams: Map[String, String]): Unit
```

`validateBatchOptions` gets the desired [KafkaOffsetRangeLimit](#) for the `startingoffsets` option in the input `caseInsensitiveParams` and with [EarliestOffsetRangeLimit](#) as the default `KafkaOffsetRangeLimit`.

`validateBatchOptions` then matches the returned [KafkaOffsetRangeLimit](#) as follows:

1. `EarliestOffsetRangeLimit` is acceptable and `validateBatchOptions` simply does nothing
2. `LatestOffsetRangeLimit` is not acceptable and `validateBatchOptions` throws an `IllegalArgumentException` :

starting offset can't be latest for batch queries on Kafka
3. `SpecificOffsetRangeLimit` is acceptable unless one of the offsets is `-1L` for which `validateBatchOptions` throws an `IllegalArgumentException` :

startingOffsets for [tp] can't be latest for batch queries on Kafka

Note

`validateBatchOptions` is used exclusively when `KafkaSourceProvider` is requested to [create a BaseRelation](#) (as a `RelationProvider`).

Writing DataFrame to Kafka Topic — `createRelation` Method (from `CreatableRelationProvider`)

```
createRelation(
  sqlContext: SQLContext,
  mode: SaveMode,
  parameters: Map[String, String],
  df: DataFrame): BaseRelation
```

Note

`createRelation` is part of the [CreatableRelationProvider Contract](#) to write the rows of a structured query (a DataFrame) to an external data source.

`createRelation` gets the `topic` option from the input `parameters`.

`createRelation` gets the [Kafka-specific options for writing](#) from the input `parameters`.

`createRelation` then uses the `KafkaWriter` helper object to [write the rows of the DataFrame to the Kafka topic](#).

In the end, `createRelation` creates a fake `BaseRelation` that simply throws an `UnsupportedOperationException` for all its methods.

`createRelation` supports `Append` and `ErrorIfExists` only. `createRelation` throws an `AnalysisException` for the other save modes:

```
Save mode [mode] not allowed for Kafka. Allowed save modes are [Append] and [ErrorIfExists] (default).
```

sourceSchema Method

```
sourceSchema(  
    sqlContext: SQLContext,  
    schema: Option[StructType],  
    providerName: String,  
    parameters: Map[String, String]): (String, StructType)
```

sourceSchema ...FIXME

```
val fromKafka = spark.read.format("kafka")...  
scala> fromKafka.printSchema  
root  
|-- key: binary (nullable = true)  
|-- value: binary (nullable = true)  
|-- topic: string (nullable = true)  
|-- partition: integer (nullable = true)  
|-- offset: long (nullable = true)  
|-- timestamp: timestamp (nullable = true)  
|-- timestampType: integer (nullable = true)
```

Note `sourceSchema` is part of Structured Streaming's `StreamSourceProvider` Contract.

Getting Desired KafkaOffsetRangeLimit (for Offset Option) — `getKafkaOffsetRangeLimit` Object Method

```
getKafkaOffsetRangeLimit(  
    params: Map[String, String],  
    offsetOptionKey: String,  
    defaultOffsets: KafkaOffsetRangeLimit): KafkaOffsetRangeLimit
```

`getKafkaOffsetRangeLimit` tries to find the given `offsetOptionKey` in the input `params` and converts the value found to a `KafkaOffsetRangeLimit` as follows:

- `latest` becomes `LatestOffsetRangeLimit`
- `earliest` becomes `EarliestOffsetRangeLimit`
- For a JSON text, `getKafkaOffsetRangeLimit` uses the `JsonUtils` helper object to read per-TopicPartition offsets from it and creates a `SpecificOffsetRangeLimit`

When the input `offsetOptionKey` was not found, `getKafkaOffsetRangeLimit` returns the input `defaultOffsets`.

Note

- `getKafkaOffsetRangeLimit` is used when:
- `KafkaSourceProvider` is requested to [validate Kafka options \(for batch queries\)](#) and [create a BaseRelation](#) (as a [RelationProvider](#))
 - (Spark Structured Streaming) `KafkaSourceProvider` is requested to `createSource` and `createContinuousReader`

Getting ConsumerStrategy per Subscription Strategy Option — `strategy` Internal Method

```
strategy(caseInsensitiveParams: Map[String, String]): ConsumerStrategy
```

`strategy` finds one of the strategy options: [subscribe](#), [subscribepattern](#) and [assign](#).

For [assign](#), `strategy` uses the `JsonUtils` helper object to [deserialize TopicPartitions from JSON](#) (e.g. `{"topicA": [0, 1], "topicB": [0, 1]}`) and returns a new [AssignStrategy](#).

For [subscribe](#), `strategy` splits the value by `,` (comma) and returns a new [SubscribeStrategy](#).

For [subscribepattern](#), `strategy` returns a new [SubscribePatternStrategy](#)

Note

- `strategy` is used when:
- `KafkaSourceProvider` is requested to [create a BaseRelation](#) (as a [RelationProvider](#))
 - (Spark Structured Streaming) `KafkaSourceProvider` is requested to `createSource` and `createContinuousReader`

`failOnDataLoss` Internal Method

```
failOnDataLoss(caseInsensitiveParams: Map[String, String]): Boolean
```

`failOnDataLoss` ...FIXME

Note

`failOnDataLoss` is used when `KafkaSourceProvider` is requested to [create a BaseRelation](#) (and also in `createSource` and `createContinuousReader` for Spark Structured Streaming).

Setting Kafka Configuration Parameters for Driver — `kafkaParamsForDriver` Object Method

```
kafkaParamsForDriver(specifiedKafkaParams: Map[String, String]): java.util.Map[String, Object]
```



`kafkaParamsForDriver` simply sets the [additional Kafka configuration parameters](#) for the driver.

Table 1. Driver's Kafka Configuration Parameters

Name	Value
key.deserializer	org.apache.kafka.common.serialization.ByteArrayDeserializer
value.deserializer	org.apache.kafka.common.serialization.ByteArrayDeserializer
auto.offset.reset	earliest

enable.auto.commit	false
max.poll.records	1
receive.buffer.bytes	65536

Tip	Enable <code>DEBUG</code> logging level for <code>org.apache.spark.sql.kafka010.KafkaSourceProvider.ConfigUpdater</code> logger to see updates of Kafka configuration parameters. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.sql.kafka010.KafkaSourceProvider.ConfigUpdater=DEB</code>
	Refer to Logging .

Note	<code>kafkaParamsForDriver</code> is used when:
	<ul style="list-style-type: none"> <code>KafkaRelation</code> is requested to build a distributed data scan with column pruning (as a <code>TableScan</code>) (Spark Structured Streaming) <code>KafkaSourceProvider</code> is requested to <code>createSource</code> and <code>createContinuousReader</code>

kafkaParamsForExecutors Object Method

```
kafkaParamsForExecutors(  
    specifiedKafkaParams: Map[String, String],  
    uniqueGroupId: String): java.util.Map[String, Object]
```

kafkaParamsForExecutors ...FIXME

Note

kafkaParamsForExecutors is used when...FIXME

kafkaParamsForProducer Object Method

```
kafkaParamsForProducer(parameters: Map[String, String]): Map[String, String]
```

kafkaParamsForProducer ...FIXME

Note

kafkaParamsForProducer is used when...FIXME

KafkaRelation

`KafkaRelation` is a [BaseRelation](#) with a [TableScan](#).

`KafkaRelation` is [created](#) exclusively when `KafkaSourceProvider` is requested to [create a BaseRelation](#) (as a [RelationProvider](#)).

`KafkaRelation` uses the fixed [schema](#).

Table 1. KafkaRelation's Schema (in the positional order)

Field Name	Data Type
<code>key</code>	<code>BinaryType</code>
<code>value</code>	<code>BinaryType</code>
<code>topic</code>	<code>StringType</code>
<code>partition</code>	<code>IntegerType</code>
<code>offset</code>	<code>LongType</code>
<code>timestamp</code>	<code>TimestampType</code>
<code>timestampType</code>	<code>IntegerType</code>

`KafkaRelation` uses the following human-readable text representation:

```
KafkaRelation(strategy=[strategy], start=[startingOffsets], end=[endingOffsets])
```

Table 2. KafkaRelation's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>pollTimeoutMs</code>	<p>Timeout (in milliseconds) to poll data from Kafka (pollTimeoutMs for <code>KafkaSourceRDD</code>)</p> <p>Initialized with the value of the following configuration properties (in the order until one found):</p> <ol style="list-style-type: none"> <code>kafkaConsumer.pollTimeoutMs</code> in the source options <code>spark.network.timeout</code> in the <code>SparkConf</code> <p>If neither is set, defaults to <code>120s</code>.</p> <p>Used exclusively when <code>KafkaRelation</code> is requested to build a distributed data scan with column pruning (and creates a <code>KafkaSourceRDD</code>).</p>

Tip Enable `INFO` or `DEBUG` logging level for `org.apache.spark.sql.kafka010.KafkaRelation` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.kafka010.KafkaRelation=DEBUG
```

Refer to [Logging](#).

Creating KafkaRelation Instance

`KafkaRelation` takes the following when created:

- `SQLContext`
- `ConsumerStrategy`
- Source options (as `Map[String, String]`) that directly correspond to the options of [DataFrameReader](#)
- User-defined Kafka parameters (as `Map[String, String]`)
- `failOnDataLoss` flag
- Starting offsets (as [KafkaOffsetRangeLimit](#))
- Ending offsets (as [KafkaOffsetRangeLimit](#))

`KafkaRelation` initializes the internal registries and counters.

Building Distributed Data Scan with Column Pruning (as TableScan) — `buildScan` Method

```
buildScan(): RDD[Row]
```

Note `buildScan` is part of [TableScan Contract](#) to build a distributed data scan with column pruning.

`buildScan` `kafkaParamsForDriver` from the [user-defined Kafka parameters](#) and uses it to create a [KafkaOffsetReader](#) (together with the [ConsumerStrategy](#), the [source options](#) and a unique group ID of the format `spark-kafka-relation-[randomUUID]-driver`).

`buildScan` then uses the `KafkaOffsetReader` to [getPartitionOffsets](#) for the starting and ending offsets and [closes](#) it right after.

`buildScan` creates a [KafkaSourceRDDOffsetRange](#) for every pair of the starting and ending offsets.

`buildScan` prints out the following INFO message to the logs:

```
GetBatch generating RDD of offset range: [comma-separated offsetRanges]
```

`buildScan` then [kafkaParamsForExecutors](#) and uses it to create a [KafkaSourceRDD](#) (with the [pollTimeoutMs](#)) and maps over all the elements (using `RDD.map` operator that creates a [MapPartitionsRDD](#)).

Tip

Use `RDD.toDebugString` to see the two RDDs, i.e. [KafkaSourceRDD](#) and [MapPartitionsRDD](#), in the RDD lineage.

In the end, `buildScan` requests the [SQLContext](#) to [create a DataFrame](#) from the [KafkaSourceRDD](#) and the [schema](#).

`buildScan` throws an `IllegalStateException` when the topic partitions for starting offsets are different from the ending offsets topics:

```
different topic partitions for starting offsets topics[[fromTopics]] and ending offset
s topics[[untilTopics]]
```

getPartitionOffsets Internal Method

```
getPartitionOffsets(
    kafkaReader: KafkaOffsetReader,
    kafkaOffsets: KafkaOffsetRangeLimit): Map[TopicPartition, Long]
```

`getPartitionOffsets` requests the input [KafkaOffsetReader](#) to [fetchTopicPartitions](#).

`getPartitionOffsets` uses the input [KafkaOffsetRangeLimit](#) to return the mapping of offsets per Kafka [TopicPartition](#) fetched:

1. For `EarliestOffsetRangeLimit`, `getPartitionOffsets` returns a map with every [TopicPartition](#) and `-2L` (as the offset)
2. For `LatestOffsetRangeLimit`, `getPartitionOffsets` returns a map with every [TopicPartition](#) and `-1L` (as the offset)
3. For `SpecificOffsetRangeLimit`, `getPartitionOffsets` returns a map from [validateTopicPartitions](#)

Note

`getPartitionOffsets` is used exclusively when `KafkaRelation` is requested to build a distributed data scan with column pruning (as a `TableScan`).

Validating TopicPartitions (Against Partition Offsets)

— `validateTopicPartitions` Inner Method

```
validateTopicPartitions(  
    partitions: Set[TopicPartition],  
    partitionOffsets: Map[TopicPartition, Long]): Map[TopicPartition, Long]
```

Note

`validateTopicPartitions` is a Scala inner method of `getPartitionOffsets`, i.e. `validateTopicPartitions` is defined within the body of `getPartitionOffsets` and so is visible and can only be used in `getPartitionOffsets`.

`validateTopicPartitions` asserts that the input set of Kafka `TopicPartitions` is exactly the set of the keys in the input `partitionOffsets`.

`validateTopicPartitions` prints out the following DEBUG message to the logs:

```
Partitions assigned to consumer: [partitions]. Seeking to [partitionOffsets]
```

In the end, `validateTopicPartitions` returns the input `partitionOffsets`.

If the input set of Kafka `TopicPartitions` is not the set of the keys in the input

`partitionOffsets`, `validateTopicPartitions` throws an `AssertionError`:

```
assertion failed: If startingOffsets contains specific offsets, you must specify all TopicPartitions.  
Use -1 for latest, -2 for earliest, if you don't care.  
Specified: [partitionOffsets] Assigned: [partitions]
```

KafkaSourceRDD

`KafkaSourceRDD` is an `RDD` of Kafka's `ConsumerRecords` (with keys and values being collections of bytes, i.e. `Array[Byte]`).

`KafkaSourceRDD` uses `KafkaSourceRDDPartition` for the `partitions`.

`KafkaSourceRDD` has a specialized API for the following RDD operators:

- `count`
- `countApprox`
- `isEmpty`
- `persist`
- `take`

`KafkaSourceRDD` is `created` when:

- `KafkaRelation` is requested to build a distributed data scan with column pruning (as a `TableScan`)
- (Spark Structured Streaming) `kafkaSource` is requested to `getBatch`

Creating KafkaSourceRDD Instance

`KafkaSourceRDD` takes the following when created:

- `SparkContext`
- Collection of key-value settings for executors reading records from Kafka topics
- Collection of `KafkaSourceRDDOffsetRanges`
- Timeout (in milliseconds) to poll data from Kafka

Used exclusively when `KafkaSourceRDD` is requested to compute a RDD partition (and requests a `KafkaDataConsumer` for a `ConsumerRecord`)

- `failOnDataLoss` flag to control...FIXME
- `reuseKafkaConsumer` flag to control...FIXME

`KafkaSourceRDD` initializes the internal registries and counters.

Computing Partition (in TaskContext) — `compute` Method

```
compute(  
    thePart: Partition,  
    context: TaskContext): Iterator[ConsumerRecord[Array[Byte], Array[Byte]]]
```

Note

`compute` is part of Spark Core's `RDD` Contract to compute a partition (in a `TaskContext`).

`compute` ...FIXME

count Operator

```
count(): Long
```

Note

`count` is part of Spark Core's `RDD` Contract to...FIXME.

`count` ...FIXME

countApprox Operator

```
countApprox(timeout: Long, confidence: Double): PartialResult[BoundedDouble]
```

Note

`countApprox` is part of Spark Core's `RDD` Contract to...FIXME.

`countApprox` ...FIXME

isEmpty Operator

```
isEmpty(): Boolean
```

Note

`isEmpty` is part of Spark Core's `RDD` Contract to...FIXME.

`isEmpty` ...FIXME

persist Operator

```
persist(newLevel: StorageLevel): this.type
```

Note	<code>persist</code> is part of Spark Core's <code>RDD</code> Contract to...FIXME.
------	--

`persist` ...FIXME

getPartitions Method

```
getPartitions: Array[Partition]
```

Note	<code>getPartitions</code> is part of Spark Core's <code>RDD</code> Contract to...FIXME
------	---

getPreferredLocations Method

```
getPreferredLocations(split: Partition): Seq[String]
```

Note	<code>getPreferredLocations</code> is part of the RDD Contract to...FIXME.
------	--

`getPreferredLocations` ...FIXME

resolveRange Internal Method

```
resolveRange(  
    consumer: KafkaDataConsumer,  
    range: KafkaSourceRDDOffsetRange): KafkaSourceRDDOffsetRange
```

`resolveRange` ...FIXME

Note	<code>resolveRange</code> is used exclusively when <code>KafkaSourceRDD</code> is requested to compute a partition.
------	---

KafkaSourceRDDOffsetRange

`KafkaSourceRDDOffsetRange` is an [offset range](#) that one `KafkaSourceRDDPartition` partition of a [KafkaSourceRDD](#) has to read.

`KafkaSourceRDDOffsetRange` is [created](#) when:

- `KafkaRelation` is requested to [build a distributed data scan with column pruning](#) (as a [TableScan](#)) (and creates a [KafkaSourceRDD](#))
- `KafkaSourceRDD` is requested to [resolveRange](#)
- (Spark Structured Streaming) `KafkaSource` is requested to `getBatch`

`KafkaSourceRDDOffsetRange` takes the following when created:

- Kafka [TopicPartition](#)
- `fromOffset`
- `untilOffset`
- Preferred location

Note

[TopicPartition](#) is a topic name and partition number.

KafkaSourceRDDPartition

KafkaSourceRDDPartition is...FIXME

ConsumerStrategy Contract — Kafka Consumer Providers

`ConsumerStrategy` is the [contract](#) for Kafka Consumer providers that can [create a Kafka Consumer](#) given Kafka parameters.

```
package org.apache.spark.sql.kafka010

sealed trait ConsumerStrategy {
  def createConsumer(kafkaParams: ju.Map[String, Object]): Consumer[Array[Byte], Array[Byte]]
}
```

Table 1. ConsumerStrategy Contract

Property	Description
<code>createConsumer</code>	Creates a Kafka Consumer (of keys and values of type <code>Array[Byte]</code>) Used exclusively when <code>KafkaOffsetReader</code> is requested to creating a Kafka Consumer

Table 2. ConsumerStrategies

ConsumerStrategy	createConsumer		
<code>AssignStrategy</code>	Uses <code>KafkaConsumer.assign(Collection<TopicPartition> partitions)</code>		
<code>SubscribeStrategy</code>	Uses <code>KafkaConsumer.subscribe(Collection<String> topics)</code>		
<code>SubscribePatternStrategy</code>	Uses <code>KafkaConsumer.subscribe(Pattern pattern, ConsumerRebalanceListener listener)</code> with <code>NoOpConsumerRebalanceListener</code> . <table border="1" style="margin-left: 20px;"> <tr> <td style="padding: 5px;">Tip</td> <td style="padding: 5px;">Refer to java.util.regex.Pattern for the format of supported topic subscription regex patterns.</td> </tr> </table>	Tip	Refer to java.util.regex.Pattern for the format of supported topic subscription regex patterns.
Tip	Refer to java.util.regex.Pattern for the format of supported topic subscription regex patterns.		

Note	<code>ConsumerStrategy</code> is a Scala sealed trait which means that all the implementations are in the same compilation unit (a single file).
------	---

KafkaOffsetReader

`KafkaOffsetReader` is used to query a Kafka cluster for partition offsets.

`KafkaOffsetReader` is [created](#) when:

- `KafkaRelation` is requested to [build a distributed data scan with column pruning](#) (as a `TableScan`) ([to get the initial partition offsets](#))
- (Spark Structured Streaming) `kafkaSourceProvider` is requested to `createSource` and `createContinuousReader`

When requested for the human-readable text representation (aka `toString`),
`KafkaOffsetReader` simply requests the [ConsumerStrategy](#) for one.

Table 1. KafkaOffsetReader's Options

Name	Default Value	Description
<code>fetchOffset.numRetries</code>	3	
<code>fetchOffset.retryIntervalMs</code>	1000	How long to wait before retries.

Table 2. KafkaOffsetReader's Internal Registries and Counters

Name	Description
consumer	<p>Kafka's Consumer (with keys and values of <code>Array[Byte]</code> type)</p> <p>Initialized when <code>KafkaOffsetReader</code> is created.</p> <p>Used when <code>KafkaOffsetReader</code> :</p> <ul style="list-style-type: none"> • fetchTopicPartitions • fetchEarliestOffsets • fetchLatestOffsets • resetConsumer • is closed
execContext	
groupId	
kafkaReaderThread	
maxOffsetFetchAttempts	
nextId	
offsetFetchAttemptIntervalMs	

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.sql.kafka010.KafkaOffsetReader</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.kafka010.KafkaOffsetReader=DEBUG</pre>
	<p>Refer to Logging.</p>

Creating Kafka Consumer — `createConsumer` Internal Method

```
createConsumer(): Consumer[Array[Byte], Array[Byte]]
```

`createConsumer` requests the [ConsumerStrategy](#) to create a Kafka Consumer with `driverKafkaParams` and `new generated group.id Kafka` property.

Note

`createConsumer` is used when `KafkaOffsetReader` is created (and initializes consumer) and `resetConsumer`

Creating KafkaOffsetReader Instance

`KafkaOffsetReader` takes the following when created:

- `ConsumerStrategy`
- Kafka parameters (as `Map[String, Object]`)
- Reader options (as `Map[String, String]`)
- Prefix for the group id

`KafkaOffsetReader` initializes the internal registries and counters.

close Method

```
close(): Unit
```

`close` ...FIXME

Note

`close` is used when...FIXME

fetchEarliestOffsets Method

```
fetchEarliestOffsets(): Map[TopicPartition, Long]
```

`fetchEarliestOffsets` ...FIXME

Note

`fetchEarliestOffsets` is used when...FIXME

fetchEarliestOffsets Method

```
fetchEarliestOffsets(newPartitions: Seq[TopicPartition]): Map[TopicPartition, Long]
```

`fetchEarliestOffsets` ...FIXME

Note

`fetchEarliestOffsets` is used when...FIXME

fetchLatestOffsets Method

```
fetchLatestOffsets(): Map[TopicPartition, Long]
```

fetchLatestOffsets ...FIXME

Note

fetchLatestOffsets is used when...FIXME

Fetching (and Pausing) Assigned Kafka TopicPartitions — fetchTopicPartitions Method

```
fetchTopicPartitions(): Set[TopicPartition]
```

fetchTopicPartitions uses an UninterruptibleThread thread to do the following:

1. Requests the Kafka Consumer to poll (fetch data) for the topics and partitions (with 0 timeout)
2. Requests the Kafka Consumer to get the set of partitions currently assigned
3. Requests the Kafka Consumer to suspend fetching from the partitions assigned

In the end, fetchTopicPartitions returns the TopicPartitions assigned (and paused).

Note

fetchTopicPartitions is used exclusively when KafkaRelation is requested to build a distributed data scan with column pruning (as a TableScan) through getPartitionOffsets.

nextGroupId Internal Method

```
nextGroupId(): String
```

nextGroupId ...FIXME

Note

nextGroupId is used when...FIXME

resetConsumer Internal Method

```
resetConsumer(): Unit
```

resetConsumer ...FIXME

Note	resetConsumer is used when...FIXME
------	------------------------------------

runUninterruptibly Internal Method

```
runUninterruptibly[T](body: => T): T
```

runUninterruptibly ...FIXME

Note	runUninterruptibly is used when...FIXME
------	---

withRetriesWithoutInterrupt Internal Method

```
withRetriesWithoutInterrupt(body: => Map[TopicPartition, Long]): Map[TopicPartition, L  
ong]
```

withRetriesWithoutInterrupt ...FIXME

Note	withRetriesWithoutInterrupt is used when...FIXME
------	--

KafkaOffsetRangeLimit

`KafkaOffsetRangeLimit` is the desired [offset range limits](#) for starting, ending, and specific offsets.

Table 1. KafkaOffsetRangeLimits

KafkaOffsetRangeLimit	Description
<code>EarliestOffsetRangeLimit</code>	Bind to the earliest offset
<code>LatestOffsetRangeLimit</code>	Bind to the latest offset
<code>SpecificOffsetRangeLimit</code>	Bind to specific offsets Takes <code>partitionOffsets</code> (as <code>Map[TopicPartition, Long]</code>) when created.

`KafkaOffsetRangeLimit` is "created" (i.e. mapped to from a human-readable text representation) when `KafkaSourceProvider` is requested to [getKafkaOffsetRangeLimit](#).

`KafkaOffsetRangeLimit` defines two constants to denote offset range limits that are resolved via Kafka:

- `-1L` for the latest offset
- `-2L` for the earliest offset

Note

`KafkaOffsetRangeLimit` is a Scala **sealed trait** which means that all the [extensions](#) are in the same compilation unit (a single file).

KafkaDataConsumer Contract

`KafkaDataConsumer` is the [contract](#) for `KafkaDataConsumers` that use an `InternalKafkaConsumer` for the following:

- Getting a single `Kafka ConsumerRecord`
- Getting a single `AvailableOffsetRange`

`KafkaDataConsumer` has to be [released](#) explicitly.

```
package org.apache.spark.sql.kafka010

sealed trait KafkaDataConsumer {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def internalConsumer: InternalKafkaConsumer
    def release(): Unit
}
```

Table 1. KafkaDataConsumer Contract

Property	Description
<code>internalConsumer</code>	Used when: <ul style="list-style-type: none"> • <code>KafkaDataConsumer</code> is requested to get a single Kafka ConsumerRecord and get a single AvailableOffsetRange • <code>CachedKafkaDataConsumer</code> and <code>NonCachedKafkaDataConsumer</code> are requested to release the <code>InternalKafkaConsumer</code>
<code>release</code>	Used when: <ul style="list-style-type: none"> • <code>KafkaSourceRDD</code> is requested to compute a partition • (Spark Structured Streaming) <code>KafkaContinuousDataReader</code> is requested to close

Table 2. KafkaDataConsumers

KafkaDataConsumer	Description
<code>CachedKafkaDataConsumer</code>	
<code>NonCachedKafkaDataConsumer</code>	

Note

`KafkaDataConsumer` is a Scala **sealed trait** which means that all the [implementations](#) are in the same compilation unit (a single file).

Getting Single Kafka ConsumerRecord — `get` Method

```
get(
  offset: Long,
  untilOffset: Long,
  pollTimeoutMs: Long,
  failOnDataLoss: Boolean): ConsumerRecord[Array[Byte], Array[Byte]]
```

`get` simply requests the [InternalKafkaConsumer](#) to get a single Kafka ConsumerRecord.

Note

`get` is used when:

- `KafkaSourceRDD` is requested to [compute a partition](#)
- (Spark Structured Streaming) `KafkaContinuousDataReader` is requested to [next](#)

Getting Single AvailableOffsetRange — `getAvailableOffsetRange` Method

```
getAvailableOffsetRange(): AvailableOffsetRange
```

`getAvailableOffsetRange` simply requests the [InternalKafkaConsumer](#) to [get a single AvailableOffsetRange](#).

Note

`getAvailableOffsetRange` is used when:

- `KafkaSourceRDD` is requested to [compute a partition](#) (through [resolveRange](#))
- (Spark Structured Streaming) `KafkaContinuousDataReader` is requested to [next](#)

InternalKafkaConsumer

InternalKafkaConsumer is...FIXME

Getting Single Kafka ConsumerRecord — get Method

```
get(  
    offset: Long,  
    untilOffset: Long,  
    pollTimeoutMs: Long,  
    failOnDataLoss: Boolean): ConsumerRecord[Array[Byte], Array[Byte]]
```

get ...FIXME

Note	get is used when...FIXME
------	--------------------------

Getting Single AvailableOffsetRange — getAvailableOffsetRange Method

```
getAvailableOffsetRange(): AvailableOffsetRange
```

getAvailableOffsetRange ...FIXME

Note	getAvailableOffsetRange is used when...FIXME
------	--

KafkaWriter Helper Object—Writing Structured Queries to Kafka

`KafkaWriter` is a Scala object that is used to [write](#) the rows of a batch (or a streaming) structured query to Apache Kafka.

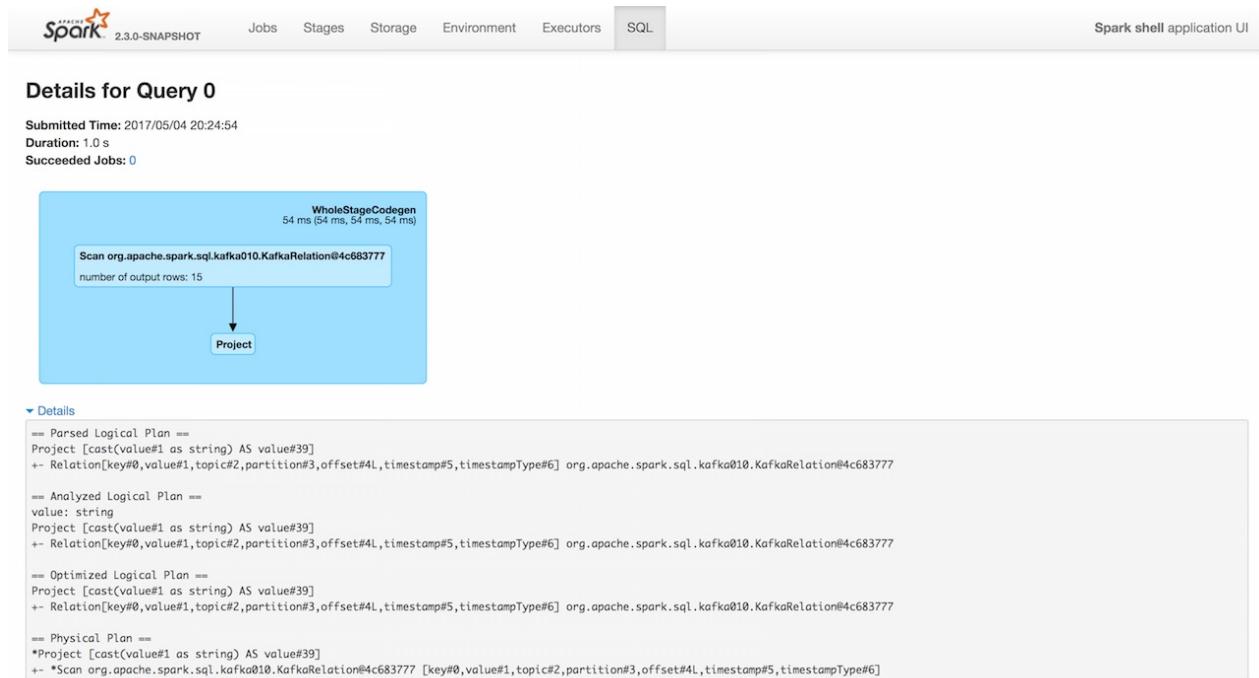


Figure 1. KafkaWriter (write) in web UI

`KafkaWriter` validates the schema of a structured query that it contains the following columns ([output schema attributes](#)):

- Either **topic** of type `StringType` or the **topic** option are defined
- Optional **key** of type `StringType` or `BinaryType`
- Required **value** of type `StringType` or `BinaryType`

```
// KafkaWriter is a private `kafka010` package object
// and so the code to use it should also be in the same package
// BEGIN: Use `:paste -raw` in spark-shell
package org.apache.spark.sql.kafka010

object PublicKafkaWriter {
    import org.apache.spark.sql.execution.QueryExecution
    def validateQuery(
        queryExecution: QueryExecution,
        kafkaParameters: Map[String, Object],
        topic: Option[String] = None): Unit = {
        import scala.collection.JavaConversions.mapAsJavaMap
        KafkaWriter.validateQuery(queryExecution, kafkaParameters, topic)
    }
}
// END

import org.apache.spark.sql.kafka010.{PublicKafkaWriter => PKW}

val spark: SparkSession = ...
val q = spark.range(1).select('id)
scala> PKW.validateQuery(
    queryExecution = q.queryExecution,
    kafkaParameters = Map.empty[String, Object])
org.apache.spark.sql.AnalysisException: topic option required when no 'topic' attribute is present. Use the topic option for setting a topic.;
  at org.apache.spark.sql.kafka010.KafkaWriter$$anonfun$2.apply(KafkaWriter.scala:53)
  at org.apache.spark.sql.kafka010.KafkaWriter$$anonfun$2.apply(KafkaWriter.scala:52)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.sql.kafka010.KafkaWriter$.validateQuery(KafkaWriter.scala:51)
  at org.apache.spark.sql.kafka010.PublicKafkaWriter$.validateQuery(<pastie>:10)
  ... 50 elided
```

Writing Rows of Structured Query to Kafka Topic

— write Method

```
write(
    sparkSession: SparkSession,
    queryExecution: QueryExecution,
    kafkaParameters: ju.Map[String, Object],
    topic: Option[String] = None): Unit
```

`write` gets the [output schema](#) of the [analyzed logical plan](#) of the input [QueryExecution](#).

`write` then validates the schema of a structured query.

In the end, `write` requests the `QueryExecution` for `RDD[InternalRow]` (that represents the structured query as an RDD) and executes the following function on every partition of the RDD (using `RDD.foreachPartition` operation):

1. Creates a `KafkaWriteTask` (for the input `kafkaParameters`, the schema and the input `topic`)
2. Requests the `KafkaWriteTask` to write the rows (of the partition) to Kafka topic
3. Requests the `KafkaWriteTask` to close

	<code>write</code> is used when:
Note	<ul style="list-style-type: none"> • <code>KafkaSourceProvider</code> is requested to write a DataFrame to a Kafka topic • (Spark Structured Streaming) <code>KafkaSink</code> is requested to <code>addBatch</code>

Validating Schema (Attributes) of Structured Query and Topic Option Availability — `validateQuery` Method

```
validateQuery(
  schema: Seq[Attribute],
  kafkaParameters: ju.Map[String, Object],
  topic: Option[String] = None): Unit
```

`validateQuery` makes sure that the following attributes are in the input schema (or their alternatives) and of the right data types:

- Either `topic` attribute of type `StringType` or the `topic` option are defined
- If `key` attribute is defined it is of type `StringType` or `BinaryType`
- `value` attribute is of type `StringType` or `BinaryType`

If any of the requirements are not met, `validateQuery` throws an `AnalysisException`.

	<code>validateQuery</code> is used when:
Note	<ul style="list-style-type: none"> • <code>KafkaWriter</code> object is requested to write the rows of a structured query to a Kafka topic • (Spark Structured Streaming) <code>KafkaStreamWriter</code> is created • (Spark Structured Streaming) <code>KafkaSourceProvider</code> is requested to <code>createStreamWriter</code>

KafkaWriteTask

`KafkaWriteTask` is used to [write rows](#) (from a structured query) to Apache Kafka.

`KafkaWriteTask` is [created](#) exclusively when `KafkaWriter` is requested to [write the rows of a structured query to a Kafka topic](#).

`KafkaWriteTask` [writes](#) keys and values in their binary format (as JVM's bytes) and so uses the [raw-memory unsafe row format](#) only (i.e. `UnsafeRow`). That is supposed to save time for reconstructing the rows to very tiny JVM objects (i.e. byte arrays).

Table 1. KafkaWriteTask's Internal Properties

Name	Description
<code>callback</code>	
<code>failedWrite</code>	
<code>projection</code>	UnsafeProjection Created once when <code>KafkaWriteTask</code> is created.

Writing Rows to Kafka Asynchronously — `execute` Method

```
execute(iterator: Iterator[InternalRow]): Unit
```

`execute` uses Apache Kafka's Producer API to create a [KafkaProducer](#) and [ProducerRecord](#) for every row in `iterator`, and sends the rows to Kafka in batches asynchronously.

Internally, `execute` creates a `KafkaProducer` using `Array[Byte]` for the keys and values, and `producerConfiguration` for the producer's configuration.

Note	<code>execute</code> creates a single <code>KafkaProducer</code> for all rows.
------	--

For every row in the `iterator`, `execute` uses the internal [UnsafeProjection](#) to *project* (aka *convert*) [binary internal row format](#) to a [UnsafeRow](#) object and take 0th, 1st and 2nd fields for a topic, key and value, respectively.

`execute` then creates a `ProducerRecord` and sends it to Kafka (using the `KafkaProducer`). `execute` registers a asynchronous `callback` to monitor the writing.

Note

From [KafkaProducer's documentation](#):

The `send()` method is asynchronous. When called it adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency.

Creating UnsafeProjection — `createProjection` Internal Method

```
createProjection: UnsafeProjection
```

`createProjection` creates a [UnsafeProjection](#) with `topic`, `key` and `value` [expressions](#) and the `inputSchema`.

`createProjection` makes sure that the following holds (and reports an `IllegalStateException` otherwise):

- `topic` was defined (either as the input `topic` or in `inputSchema`) and is of type `StringType`
- Optional `key` is of type `StringType` or `BinaryType` if defined
- `value` was defined (in `inputSchema`) and is of type `StringType` or `BinaryType`

`createProjection` casts `key` and `value` [expressions](#) to `BinaryType` in [UnsafeProjection](#).

Note

`createProjection` is used exclusively when `KafkaWriteTask` is created (as `projection`).

close Method

```
close(): Unit
```

`close` ...FIXME

Note

`close` is used when...FIXME

Creating KafkaWriteTask Instance

`KafkaWriteTask` takes the following when created:

- Kafka Producer configuration (as `Map[String, Object]`)

- Input schema (as `Seq[Attribute]`)
- Topic name

`KafkaWriteTask` initializes the [internal registries and counters](#).

JsonUtils Helper Object

`JsonUtils` is a Scala object with [methods](#) for serializing and deserializing Kafka [TopicPartitions](#) to and from a single JSON text.

`JsonUtils` uses [json4s](#) library that provides a single AST with the Jackson parser for parsing to the AST (using `json4s-jackson` module).

Table 1. JsonUtils API

Name	Description
<code>partitionOffsets</code>	Deserializing partition offsets (i.e. offsets per Kafka <code>TopicPartition</code>) from JSON, e.g. <code>{"topicA":{"0":23,"1":-1}, "topicB":{"0":-2}}</code> <code>partitionOffsets(str: String): Map[TopicPartition, Long]</code>
<code>partitionOffsets</code>	Serializing partition offsets (i.e. offsets per Kafka <code>TopicPartition</code>) to JSON <code>partitionOffsets(partitionOffsets: Map[TopicPartition, Long]): String</code>
<code>partitions</code>	Deserializing <code>TopicPartitions</code> from JSON, e.g. <code>{"topicA": [0,1], "topicB": [0,1]}</code> <code>partitions(str: String): Array[TopicPartition]</code>
<code>partitions</code>	Serializing <code>TopicPartitions</code> to JSON <code>partitions(partitions: Iterable[TopicPartition]): String</code>

Deserializing Partition Offsets From JSON — `partitionOffsets` Method

```
partitionOffsets(str: String): Map[TopicPartition, Long]
```

`partitionOffsets ...FIXME`

Note

- `partitionOffsets` is used when:
- `KafkaSourceProvider` is requested to [get the desired KafkaOffsetRangeLimit \(for offset option\)](#)
 - (Spark Structured Streaming) `KafkaContinuousReader` is requested to `deserializeOffset`
 - (Spark Structured Streaming) `KafkaSourceOffset` is created (from a `SerializedOffset`)

Serializing Partition Offsets to JSON — `partitionOffsets` Method

```
partitionOffsets(partitionOffsets: Map[TopicPartition, Long]): String
```

`partitionOffsets` ...FIXME

Note

`partitionOffsets` is used when...FIXME

Serializing TopicPartitions to JSON — `partitions` Method

```
partitions(partitions: Iterable[TopicPartition]): String
```

`partitions` ...FIXME

Note

`partitions` seems not to be used.

Deserializing TopicPartitions from JSON — `partitions` Method

```
partitions(str: String): Array[TopicPartition]
```

`partitions` uses json4s-jackson's `Serialization` object to read a `Map[String, Seq[Int]]` from the input string that represents a `Map` of topics and partition numbers, e.g. `{"topicA": [0,1], "topicB": [0,1]}`.

For every pair of topic and partition number, `partitions` creates a new Kafka [TopicPartition](#).

In case of any parsing issues, `partitions` throws a new `IllegalArgumentException`:

Expected e.g. `{"topicA": [0,1], "topicB": [0,1]}`, got [str]

Note

`partitions` is used exclusively when `KafkaSourceProvider` is requested for a [ConsumerStrategy](#) (given `assign` option).

Avro Data Source

Spark SQL supports structured queries over [Avro files](#) as well as in [columns](#) (in a `DataFrame`).

	<p>Apache Avro is a data serialization format and provides the following features:</p> <ul style="list-style-type: none"> • Language-independent (with language bindings for popular programming languages, e.g. Java, Python) • Rich data structures • A compact, fast, binary data format (encoding) • A container file for sequences of Avro data (aka <i>Avro data files</i>) • Remote procedure call (RPC) • Optional code generation (optimization) to read or write data files, and implement RPC protocols
Note	

Avro data source is provided by the `spark-avro` external module. You should include it as a dependency in your Spark application (e.g. `spark-submit --packages` or in `build.sbt`).

```
org.apache.spark:spark-avro_2.12:2.4.0
```

The following shows how to include the `spark-avro` module in a `spark-shell` session.

```
$ ./bin/spark-shell --packages org.apache.spark:spark-avro_2.12:2.4.0
```

Table 1. Functions for Avro

Name	Description
<code>from_avro</code>	<pre>from_avro(data: Column, jsonFormatSchema: String): Column</pre> <p>Parses an Avro-encoded binary column and converts to a Catalyst value per JSON-encoded Avro schema</p>
<code>to_avro</code>	<pre>to_avro(data: Column): Column</pre> <p>Converts a column to an Avro-encoded binary column</p>

After the module is loaded, you should import the `org.apache.spark.sql.avro` package to have the `from_avro` and `to_avro` functions available.

```
import org.apache.spark.sql.avro._
```

Converting Column to Avro-Encoded Binary Column — `to_avro` Method

```
to_avro(data: Column): Column
```

`to_avro` creates a `Column` with the `CatalystDataToAvro` unary expression (with the `Catalyst expression` of the given `data` column).

```
import org.apache.spark.sql.avro._
val q = spark.range(1).withColumn("to_avro_id", to_avro('id))
scala> q.show
+---+-----+
| id|to_avro_id|
+---+-----+
|  0|[00]|
+---+-----+

val logicalPlan = q.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Project [id#33L, catalystdatatoavro('id) AS to_avro_id#35]
01 +- Range (0, 1, step=1, splits=Some(8))

import org.apache.spark.sql.avro.CatalystDataToAvro
// Let's use QueryExecution.analyzed instead
// https://issues.apache.org/jira/browse/SPARK-26063
val analyzedPlan = q.queryExecution.analyzed
val toAvroExpr = analyzedPlan.expressions.drop(1).head.children.head.asInstanceOf[CatalystDataToAvro]
scala> println(toAvroExpr.sql)
to_avro(`id`, bigint)
```

Converting Avro-Encoded Column to Catalyst Value — `from_avro` Method

```
from_avro(data: Column, jsonFormatsSchema: String): Column
```

`from_avro` creates a [Column](#) with the [AvroDataToCatalyst](#) unary expression (with the [Catalyst expression](#) of the given `data` column and the `jsonFormatSchema` JSON-encoded schema).

```
import org.apache.spark.sql.avro._

val data = spark.range(1).withColumn("to_avro_id", to_avro('id))

// Use from_avro to decode to_avro-encoded id column
val jsonFormatSchema = """
  |{
  |  "type": "long",
  |  "name": "id"
  |}
"""

scala> q.show
+-----+
|id_from_avro|
+-----+
|          0|
+-----+


val logicalPlan = q.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Project [avrodatatocatalyst('to_avro_id,
01 {
02   "type": "long",
03   "name": "id"
04 }
05 ) AS id_from_avro#77]
06 +- Project [id#66L, catalyststdatatoavro(id#66L) AS to_avro_id#68]
07   +- Range (0, 1, step=1, splits=Some(8))

import org.apache.spark.sql.avro.AvroDataToCatalyst
// Let's use QueryExecution.analyzed instead
// https://issues.apache.org/jira/browse/SPARK-26063
val analyzedPlan = q.queryExecution.analyzed
val fromAvroExpr = analyzedPlan.expressions.head.children.head.asInstanceOf[AvroDataToCatalyst]
scala> println(fromAvroExpr.sql)
from_avro(`to_avro_id`, bigint)
```

AvroFileFormat — FileFormat For Avro-Encoded Files

`AvroFileFormat` is a [FileFormat](#) for Apache Avro, i.e. a data source format that can read and write Avro-encoded data in files.

`AvroFileFormat` is a [DataSourceRegister](#) and registers itself as **avro** data source.

```
// ./bin/spark-shell --packages org.apache.spark:spark-avro_2.12:2.4.0

// Writing data to Avro file(s)
spark
  .range(1)
  .write
  .format("avro") // <-- Triggers AvroFileFormat
  .save("data.avro")

// Reading Avro data from file(s)
val q = spark
  .read
  .format("avro") // <-- Triggers AvroFileFormat
  .load("data.avro")
scala> q.show
+---+
| id|
+---+
|  0|
+---+
```

`AvroFileFormat` is [splittable](#), i.e. `FIXME`

Building Partitioned Data Reader — `buildReader` Method

```
buildReader(
  spark: SparkSession,
  dataSchema: StructType,
  partitionSchema: StructType,
  requiredSchema: StructType,
  filters: Seq[Filter],
  options: Map[String, String],
  hadoopConf: Configuration): (PartitionedFile) => Iterator[InternalRow]
```

Note	<code>buildReader</code> is part of the FileFormat Contract to build a PartitionedFile reader.
------	--

```
buildReader ...FIXME
```

Inferring Schema — `inferSchema` Method

```
inferSchema(  
    spark: SparkSession,  
    options: Map[String, String],  
    files: Seq[FileStatus]): Option[StructType]
```

Note	<code>inferSchema</code> is part of the FileFormat Contract to infer (return) the <code>schema</code> of the given files.
------	---

```
inferSchema ...FIXME
```

Preparing Write Job — `prepareWrite` Method

```
prepareWrite(  
    spark: SparkSession,  
    job: Job,  
    options: Map[String, String],  
    dataSchema: StructType): OutputWriterFactory
```

Note	<code>preparewrite</code> is part of the FileFormat Contract to prepare a write job.
------	--

```
prepareWrite ...FIXME
```

AvroOptions — Avro Data Source Options

`AvroOptions` represents the [options](#) of the [Avro data source](#).

Table 1. Options for Avro Data Source

Option / Key	Default Value	Description
<code>avroSchema</code>	(undefined)	Avro schema in JSON format
<code>compression</code>	(undefined)	<p>Specifies the compression codec to use when writing data to disk</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Note If the option is not defined explicitly, Avro data source uses spark.sql.avro.compression.codec configuration property. </div>
<code>ignoreExtension</code>	<code>false</code>	<p>Controls whether Avro data source should read all files regardless of their extension (<code>true</code>) or not (<code>false</code>).</p> <p>By default, Avro data source reads only files with file extension.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Note If the option is not defined explicitly, Avro data source uses avro.mapred.ignore.inputs.without.extension Hadoop runtime property. </div>
<code>recordName</code>	<code>topLevelRecord</code>	<p>Top-level record name when writing Avro data to disk.</p> <p>Consult Apache Avro™ 1.8.2 Specification</p>
<code>recordNamespace</code>	(empty)	<p>Record namespace when writing Avro data to disk.</p> <p>Consult Apache Avro™ 1.8.2 Specification</p>
Note		The options are case-insensitive.

`AvroOptions` is created when `AvroFileFormat` is requested to `inferSchema`, `prepareWrite` and `buildReader`.

Creating AvroOptions Instance

`AvroOptions` takes the following when created:

- Case-insensitive configuration parameters (i.e. `Map[String, String]`)

- Hadoop Configuration

CatalystDataToAvro Unary Expression

`CatalystDataToAvro` is a [unary expression](#) that represents `to_avro` function in a structured query.

`CatalystDataToAvro` takes a single [Catalyst expression](#) when created.

`CatalystDataToAvro` generates Java source code (as `ExprCode`) for code-generated expression evaluation.

```
import org.apache.spark.sql.avro.CatalystDataToAvro
val catalystDataToAvro = CatalystDataToAvro($"id".expr)

import org.apache.spark.sql.catalyst.expressions.codegen.{CodegenContext, ExprCode}
val ctx = new CodegenContext
// doGenCode is used when Expression.genCode is executed
// FIXME The following won't work due to https://issues.apache.org/jira/browse/SPARK-2
6063
val ExprCode(code, _, _) = catalystDataToAvro.genCode(ctx)

// Helper methods
def trim(code: String): String = {
  code.trim.split("\n").map(_.trim).filter(line => line.nonEmpty).mkString("\n")
}
def prettyPrint(code: String) = println(trim(code))
// END: Helper methods

scala> println(trim(code))
// FIXME: Finish me once https://issues.apache.org/jira/browse/SPARK-26063 is fixed
// See the following example
```

```

// Let's use a workaround to create a CatalystDataToAvro expression
// with the child resolved
val q = spark.range(1).withColumn("to_avro_id", to_avro('id))
import org.apache.spark.sql.avro.CatalystDataToAvro
val analyzedPlan = q.queryExecution.analyzed
val catalystDataToAvro = analyzedPlan.expressions.drop(1).head.children.head.asInstanceOf[CatalystDataToAvro]

import org.apache.spark.sql.catalyst.expressions.codegen.{CodegenContext, ExprCode}
val ctx = new CodegenContext
val ExprCode(code, _, _) = catalystDataToAvro.genCode(ctx)

// Doh! It does not work either
// java.lang.UnsupportedOperationException: Cannot evaluate expression: id#38L

// Let's try something else (more serious)

import org.apache.spark.sql.catalyst.expressions.{BindReferences, Expression}
val boundExprs = analyzedPlan.expressions.map { e =>
  BindReferences.bindReference[Expression](e, analyzedPlan.children.head.output)
}
// That should trigger doGenCode
val codes = ctx.generateExpressions(boundExprs)

// The following corresponds to catalystDataToAvro.genCode(ctx)
val ExprCode(code, _, _) = codes.tail.head

// Helper methods
def trim(code: String): String = {
  code.trim.split("\n").map(_.trim).filter(line => line.nonEmpty).mkString("\n")
}
def prettyPrint(code: String) = println(trim(code))
// END: Helper methods

scala> println(trim(code.toString))
long value_7 = i.getLong(0);
byte[] value_6 = null;
value_6 = (byte[]) ((org.apache.spark.sql.avro.CatalystDataToAvro) references[2] /* this */).nullSafeEval(value_7);

```

Generating Java Source Code (ExprCode) For Code-Generated Expression Evaluation — doGenCode Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

`doGenCode` is part of [Expression Contract](#) to generate a Java source code (`ExprCode`) for code-generated expression evaluation.

`doGenCode` requests the `CodegenContext` to generate code to reference this [CatalystDataToAvro](#) instance.

In the end, `doGenCode` `defineCodeGen` with the function `f` that uses `nullSafeEval`.

nullSafeEval Method

```
nullSafeEval(input: Any): Any
```

Note

`nullSafeEval` is part of the [UnaryExpression Contract](#) to...FIXME.

`nullSafeEval` ...FIXME

AvroDataToCatalyst Unary Expression

`AvroDataToCatalyst` is a [unary expression](#) that represents `from_avro` function in a structured query.

`AvroDataToCatalyst` takes the following when created:

- [Catalyst expression](#)
- JSON-encoded Avro schema

`AvroDataToCatalyst` generates Java source code (as `ExprCode`) for code-generated expression evaluation.

Generating Java Source Code (`ExprCode`) For Code-Generated Expression Evaluation — `doGenCode` Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note	<code>doGenCode</code> is part of Expression Contract to generate a Java source code (<code>ExprCode</code>) for code-generated expression evaluation.
------	--

`doGenCode` requests the `CodegenContext` to [generate code to reference this AvroDataToCatalyst instance](#).

In the end, `doGenCode` [defineCodeGen](#) with the function `f` that uses `nullSafeEval`.

nullSafeEval Method

```
nullSafeEval(input: Any): Any
```

Note	<code>nullSafeEval</code> is part of the UnaryExpression Contract to...FIXME.
------	---

`nullSafeEval` ...FIXME

JDBC Data Source

Spark SQL supports loading data from tables using JDBC.

JDBC

The **JDBC API** is the Java™ SE standard for database-independent connectivity between the Java™ programming language and a wide range of databases: SQL or NoSQL databases and tabular data sources like spreadsheets or flat files.

Read more on the JDBC API in [JDBC Overview](#) and in the official Java SE 8 documentation in [Java JDBC API](#).

As a Spark developer, you use [DataFrameReader.jdbc](#) to load data from an external table using JDBC.

```
val table = spark.read.jdbc(url, table, properties)

// Alternatively
val table = spark.read.format("jdbc").options(...).load(...)
```

These one-liners create a [DataFrame](#) that represents the distributed process of loading data from a database and a table (with additional properties).

JDBCOptions — JDBC Data Source Options

`JDBCOptions` represents the options of the [JDBC data source](#).

Table 1. Options for JDBC Data Source

Option / Key	Default Value	Description
<code>batchsize</code>	1000	The minimum value is 1 Used exclusively when JdbcRelationProvider.saveTable a structured query (a DataFrame) to a table.
<code>createTableColumnTypes</code>		Used exclusively when JdbcRelationProvider.createTable a structured query (a DataFrame) to a table.
<code>createTableOptions</code>	Empty string	Used exclusively when JdbcRelationProvider.createTable a structured query (a DataFrame) to a table.
<code>customSchema</code>	(undefined)	<p>Specifies the custom data types of the relation.</p> <p><code>customSchema</code> is a comma-separated list of column names and their data types in a canonical SQL representation, such as <code>name1 STRING, name2 INT</code>. <code>customSchema</code> defines the data types of the columns if they are inferred from the table schema and follow the rules defined below.</p> <pre> colTypeList : colType (',' colType)* ; colType : identifier dataType (COMMENT STRING)? ; dataType : complex=ARRAY '<' dataType '>'+ complex=MAP '<' dataType ',' dataType '>'+ complex=STRUCT ('<' complexColType ',' complexColType '>'+ identifier '('(' INTEGER_VALUE (identifier ')')+ ; </pre> <p>Used exclusively when JDBCRelation is used.</p>
		(required) Used when:

		<ul style="list-style-type: none"> <code>JDBCRDD</code> is requested to <code>resolveTableSchema</code> and <code>compute a partition</code> <code>JDBCRelation</code> is requested to <code>insert text representation</code> <code>JdbcRelationProvider</code> is requested <code>DataFrame</code> to a table <code>JdbcUtils</code> is requested to <code>tableExists</code>, <code>saveTable</code> and <code>createTable</code> <code>JDBCOptions</code> is <code>created</code> (with the input options) <code>DataFrameReader</code> is requested to <code>load source</code> (using <code>DataFrameReader.jdbc</code> and <code>dbtable</code> options)
<code>dbtable</code>		<p>(recommended) Class name of the JDE</p> <p>Used exclusively when <code>JDBCoptions</code> is <code>None</code>. In this case the JDBC driver class will get registered.</p>
<code>driver</code>		<p><code>Note</code> <code>driver</code> takes precedence over <code>options</code>.</p> <p>After the JDBC driver class was registered, <code>JdbcUtils</code> helper object is requested to</p>
<code>fetchsize</code>	0	<p>Hint to the JDBC driver as to the number of rows to fetch from the database when more rows are needed for a Statement</p> <p>The minimum value is 0 (which tells the driver to fetch all rows).</p> <p>Used exclusively when <code>JDBCRDD</code> is required.</p>
<code>isolationLevel</code>	READ_UNCOMMITTED	<p>One of the following:</p> <ul style="list-style-type: none"> NONE READ_UNCOMMITTED READ_COMMITTED REPEATABLE_READ SERIALIZABLE <p>Used exclusively when <code>JdbcUtils</code> is required.</p>
<code>lowerBound</code>		<p>Lower bound of partition column</p> <p>Used exclusively when <code>JdbcRelationProvider</code> is requested for reading.</p>

<code>numPartitions</code>		Number of partitions to use for loading o Used when: <ul style="list-style-type: none"><code>JdbcRelationProvider</code> is requested<code>JdbcUtils</code> is requested to <code>saveTab</code>
<code>partitionColumn</code>		Name of the column used to partition da Used exclusively when <code>JdbcRelationProv</code> reading (with proper <code>JDBCPartitions</code> wit When defined, the <code>lowerBound</code> , <code>upperBc</code> required. When undefined, <code>lowerBound</code> and <code>upper</code>
<code>truncate</code>	<code>false</code>	(used only for writing) Enables table trun Used exclusively when <code>JdbcRelationProv</code> structured query (a <code>DataFrame</code>) to a tabl
<code>sessionInitStatement</code>		A generic SQL statement (or PL/SQL blc Used exclusively when <code>JDBCRDD</code> is requi
<code>upperBound</code>		Upper bound of the partition column Used exclusively when <code>JdbcRelationProv</code> reading
<code>url</code>		(required) A JDBC URL to use to conne

Note	The <code>options</code> are case-insensitive.
------	--

`JDBCOptions` is `created` when:

- `DataFrameReader` is requested to load data from an external table using `JDBC` (and create a `DataFrame` to represent the process of loading the data)
- `JdbcRelationProvider` is requested to create a `BaseRelation` (as a `RelationProvider` for loading and a `CreatableRelationProvider` for writing)

Creating JDBCOptions Instance

`JDBCOptions` takes the following when created:

- JDBC URL
- Name of the table

- Case-insensitive configuration parameters (i.e. `Map[String, String]`)

The input `URL` and `table` are set as the current `url` and `dbtable` options (overriding the values in the input `parameters` if defined).

Converting Parameters (Options) to Java Properties — `asProperties` Property

```
asProperties: Properties
```

`asProperties` ...FIXME

Note

`asProperties` is used when:

- `JDBCRDD` is requested to `compute a partition` (that requests a `JdbcDialect` to `beforeFetch`)
- `JDBCRelation` is requested to `insert a data` (from a `DataFrame`) to a table

asConnectionProperties Property

```
asConnectionProperties: Properties
```

`asConnectionProperties` ...FIXME

Note

`asConnectionProperties` is used exclusively when `JdbcUtils` is requested to `createConnectionFactory`

JdbcRelationProvider

`JdbcRelationProvider` is a [DataSourceRegister](#) and registers itself to handle **jdbc** data source format.

Note	<code>JdbcRelationProvider</code> uses META-INF/services/org.apache.spark.sql.sources.DataSourceRegister file for the registration which is available in the source code of Apache Spark.
-------------	---

`JdbcRelationProvider` is a [RelationProvider](#) and a [CreatableRelationProvider](#).

`JdbcRelationProvider` is used when `DataFrameReader` is requested to load data from [jdbc](#) data source.

```
val table = spark.read.jdbc(...)

// or in a more verbose way
val table = spark.read.format("jdbc").load(...)
```

Loading Data from Table Using JDBC — `createRelation` Method (from RelationProvider)

```
createRelation(  
    sqlContext: SQLContext,  
    parameters: Map[String, String]): BaseRelation
```

Note	<code>createRelation</code> is part of RelationProvider Contract to create a BaseRelation for reading.
-------------	--

`createRelation` creates a `JDBCPartitioningInfo` (using [JDBCOptions](#) and the input parameters that correspond to the [Options for JDBC Data Source](#)).

Note	<code>createRelation</code> uses partitionColumn , lowerBound , upperBound and numPartitions .
-------------	--

In the end, `createRelation` creates a [JDBCRelation](#) with [column partitions](#) (and [JDBCOptions](#)).

Writing Rows of Structured Query (DataFrame) to Table Using JDBC — `createRelation` Method (from CreatableRelationProvider)

```
createRelation(  
    sqlContext: SQLContext,  
    mode: SaveMode,  
    parameters: Map[String, String],  
    df: DataFrame): BaseRelation
```

Note

`createRelation` is part of the [CreatableRelationProvider Contract](#) to write the rows of a structured query (a DataFrame) to an external data source.

Internally, `createRelation` creates a [JDBCOptions](#) (from the input `parameters`).

`createRelation` reads [caseSensitiveAnalysis](#) (using the input `sqlContext`).

`createRelation` checks whether the table (given `dbtable` and `url` `options` in the input `parameters`) exists.

Note

`createRelation` uses a database-specific `JdbcDialect` to check whether a table exists.

`createRelation` branches off per whether the table already exists in the database or not.

If the table **does not** exist, `createRelation` creates the table (by executing `CREATE TABLE` with [createTableColumnTypes](#) and [createTableOptions](#) options from the input `parameters`) and writes the rows to the database in a single transaction.

If however the table **does** exist, `createRelation` branches off per [SaveMode](#) (see the following [createRelation and SaveMode](#)).

Table 1. `createRelation` and `SaveMode`

Name	Description
Append	Saves the records to the table.
ErrorIfExists	Throws a <code>AnalysisException</code> with the message: <code>Table or view '[table]' already exists. SaveMode: ErrorIfExists.</code>
Ignore	Does nothing.
Overwrite	Truncates or drops the table <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Note <code>createRelation</code> truncates the table only when <code>truncate</code> JDBC option is enabled and <code>isCascadingTruncateTable</code> is disabled. </div>

In the end, `createRelation` closes the JDBC connection to the database and creates a [JDBCRelation](#).

JDBCRelation — Relation with Inserting or Overwriting Data, Column Pruning and Filter Pushdown

`JDBCRelation` is a `BaseRelation` that supports `inserting or overwriting data` and `column pruning with filter pushdown`.

As a `BaseRelation`, `JDBCRelation` defines the `schema of tuples (data)` and the `SQLContext`.

As a `InsertableRelation`, `JDBCRelation` supports `inserting or overwriting data`.

As a `PrunedFilteredScan`, `JDBCRelation` supports `building distributed data scan with column pruning and filter pushdown`.

`JDBCRelation` is `created` when:

- `DataFrameReader` is requested to load data from an external table using JDBC data source
- `JdbcRelationProvider` is requested to create a `BaseRelation` for reading data from a JDBC table

When requested for a human-friendly text representation, `JDBCRelation` requests the `JDBCOptions` for the name of the table and the `number of partitions` (if defined).

```
JDBCRelation([table]) [numPartitions=[number]]
```

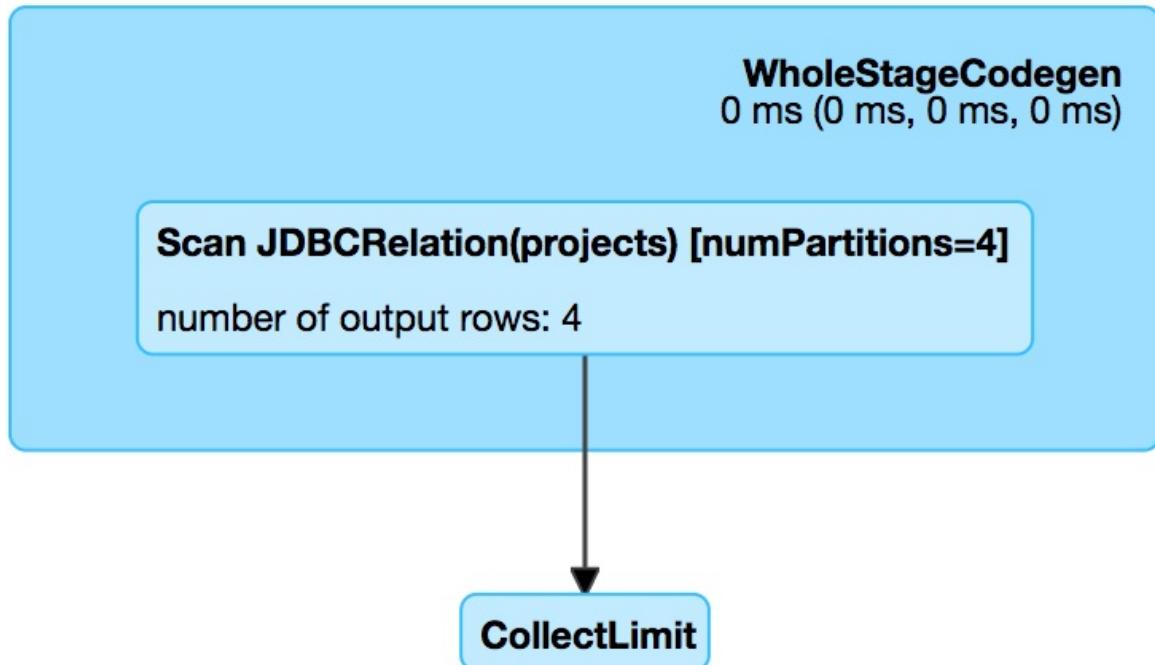


Figure 1. JDBCRelation in web UI (Details for Query)

```

scala> df.explain
== Physical Plan ==
*Scan JDBCRelation(projects) [numPartitions=1] [id#0, name#1, website#2] ReadSchema: str
uct<id:int, name:string, website:string>
  
```

`JDBCRelation` uses the [SparkSession](#) to return a [SQLContext](#).

`JDBCRelation` turns the [needConversion](#) flag off (to announce that [buildScan](#) returns an `RDD[InternalRow]` already and `DataSourceStrategy` execution planning strategy does not have to do the [RDD conversion](#)).

Creating JDBCRelation Instance

`JDBCRelation` takes the following when created:

- Array of Spark Core's `Partitions`
- `JDBCOptions`
- `SparkSession`

Finding Unhandled Filter Predicates

— `unhandledFilters` Method

```
unhandledFilters(filters: Array[Filter]): Array[Filter]
```

Note

`unhandledFilters` is part of [BaseRelation Contract](#) to find unhandled [Filter predicates](#).

`unhandledFilters` returns the [Filter predicates](#) in the input `filters` that could not be converted to a [SQL expression](#) (and are therefore unhandled by the JDBC data source natively).

Schema of Tuples (Data) — `schema` Property

`schema: StructType`

Note

`schema` is part of [BaseRelation Contract](#) to return the [schema](#) of the tuples in a relation.

`schema` uses `JDBCRDD` to [resolveTable](#) given the [JDBCOptions](#) (that simply returns the [Catalyst schema](#) of the table, also known as the default table schema).

If `customSchema` JDBC option was defined, `schema` uses `JdbcUtils` to [replace the data types in the default table schema](#).

Inserting or Overwriting Data to JDBC Table — `insert` Method

`insert(data: DataFrame, overwrite: Boolean): Unit`

Note

`insert` is part of [InsertableRelation Contract](#) that inserts or overwrites data in a relation.

`insert` simply requests the input `DataFrame` for a [DataFrameWriter](#) that in turn is requested to [save the data to a table using the JDBC data source](#) (itself!) with the `url`, `table` and `all options`.

`insert` also requests the `DataFrameWriter` to [set the save mode](#) as `Overwrite` or `Append` per the input `overwrite` flag.

Note

`insert` uses a "trick" to reuse a code that is responsible for [saving data to a JDBC table](#).

Building Distributed Data Scan with Column Pruning and Filter Pushdown — `buildScan` Method

```
buildScan(requiredColumns: Array[String], filters: Array[Filter]): RDD[Row]
```

Note	<p><code>buildScan</code> is part of PrunedFilteredScan Contract to build a distributed data scan (as a <code>RDD[Row]</code>) with support for column pruning and filter pushdown.</p>
------	---

`buildScan` uses the `JDBCRDD` object to [create a `RDD\[Row\]` for a distributed data scan](#).

JDBCRDD

`JDBCRDD` is a `RDD` of [internal binary rows](#) that represents a structured query over a table in a database accessed via JDBC.

Note	<code>JDBCRDD</code> represents a "SELECT requiredColumns FROM table" query.
------	--

`JDBCRDD` is [created](#) exclusively when `JDBCRDD` is requested to [scanTable](#) (when `JDBCRelation` is requested to [build a scan](#)).

Table 1. JDBCRDD's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>columnList</code>	Column names Used when...FIXME
<code>filterWhereClause</code>	Filters as a SQL <code>WHERE</code> clause Used when...FIXME

Computing Partition (in TaskContext) — `compute` Method

```
compute(thePart: Partition, context: TaskContext): Iterator[InternalRow]
```

Note	<code>compute</code> is part of Spark Core's <code>RDD</code> Contract to compute a partition (in a <code>TaskContext</code>).
------	---

`compute` ...FIXME

`resolveTable` Method

```
resolveTable(options: JDBCOptions): StructType
```

`resolveTable` ...FIXME

Note	<code>resolveTable</code> is used exclusively when <code>JDBCRelation</code> is requested for the schema .
------	--

Creating RDD for Distributed Data Scan — `scanTable` Object Method

```
scanTable(
    sc: SparkContext,
    schema: StructType,
    requiredColumns: Array[String],
    filters: Array[Filter],
    parts: Array[Partition],
    options: JDBCOptions): RDD[InternalRow]
```

`scanTable` takes the [url](#) option.

`scanTable` finds the corresponding JDBC dialect (per the `url` option) and requests it to quote the column identifiers in the input `requiredColumns`.

`scanTable` uses the `JdbcUtils` object to [createConnectionFactory](#) and [prune columns](#) from the input `schema` to include the input `requiredColumns` only.

In the end, `scanTable` creates a new [JDBCRDD](#).

Note

`scanTable` is used exclusively when `JDBCRelation` is requested to [build a distributed data scan with column pruning and filter pushdown](#).

Creating JDBCRDD Instance

`JDBCRDD` takes the following when created:

- `SparkContext`
- Function to create a `Connection` (`() => Connection`)
- Schema ([StructType](#))
- Array of column names
- Array of [Filter predicates](#)
- Array of Spark Core's `Partitions`
- Connection URL
- [JDBCOptions](#)

`JDBCRDD` initializes the [internal registries and counters](#).

getPartitions Method

```
getPartitions: Array[Partition]
```

Note`getPartitions` is part of Spark Core's `RDD` Contract to...FIXME`getPartitions` simply returns the `partitions` (this `JDBCRDD` was created with).

pruneSchema Internal Method

`pruneSchema(schema: StructType, columns: Array[String]): StructType``pruneSchema` ...FIXME**Note**`pruneSchema` is used when...FIXME

Converting Filter Predicate to SQL Expression — compileFilter Object Method

`compileFilter(f: Filter, dialect: JdbcDialect): Option[String]``compileFilter` ...FIXME**Note**`compileFilter` is used when:

- `JDBCRelation` is requested to [find unhandled Filter predicates](#)
- `JDBCRDD` is [created](#)

JdbcDialect

`JdbcDialect` is the [base](#) of [JDBC dialects](#) that [handle a specific JDBC URL](#) (and handle necessary type-related conversions to properly load a data from a table into a `DataFrame`).

```
package org.apache.spark.sql.jdbc

abstract class JdbcDialect extends Serializable {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def canHandle(url : String): Boolean
}
```

Table 1. (Subset of) JdbcDialect Contract

Property	Description
<code>canHandle</code>	Used when...FIXME

Table 2. JdbcDialects

JdbcDialect	Description
<code>AggregatedDialect</code>	
<code>DB2Dialect</code>	
<code>DerbyDialect</code>	
<code>MsSqlServerDialect</code>	
<code>MySQLDialect</code>	
<code>NoopDialect</code>	
<code>OracleDialect</code>	
<code>PostgresDialect</code>	
<code>TeradataDialect</code>	

getCatalystType Method

```
getCatalystType(  
    sqlType: Int,  
    typeName: String,  
    size: Int,  
    md: MetadataBuilder): Option[DataType]
```

getCatalystType ...FIXME

Note

getCatalystType is used when...FIXME

getJDBCType Method

```
getJDBCType(dt: DataType): Option[JdbcType]
```

getJDBCType ...FIXME

Note

getJDBCType is used when...FIXME

quoteIdentifier Method

```
quoteIdentifier(colName: String): String
```

quoteIdentifier ...FIXME

Note

quoteIdentifier is used when...FIXME

getTableExistsQuery Method

```
getTableExistsQuery(table: String): String
```

getTableExistsQuery ...FIXME

Note

getTableExistsQuery is used when...FIXME

getSchemaQuery Method

```
getSchemaQuery(table: String): String
```

getSchemaQuery ...FIXME

Note	getSchemaQuery is used when...FIXME
------	-------------------------------------

getTruncateQuery Method

```
getTruncateQuery(table: String): String
```

getTruncateQuery ...FIXME

Note	getTruncateQuery is used when...FIXME
------	---------------------------------------

beforeFetch Method

```
beforeFetch(connection: Connection, properties: Map[String, String]): Unit
```

beforeFetch ...FIXME

Note	beforeFetch is used when...FIXME
------	----------------------------------

escapeSql Internal Method

```
escapeSql(value: String): String
```

escapeSql ...FIXME

Note	escapeSql is used when...FIXME
------	--------------------------------

compileValue Method

```
compileValue(value: Any): Any
```

compileValue ...FIXME

Note	compileValue is used when...FIXME
------	-----------------------------------

isCascadingTruncateTable Method

```
isCascadingTruncateTable(): Option[Boolean]
```

`isCascadingTruncateTable ...FIXME`

Note

`isCascadingTruncateTable` is used when...FIXME

JdbcUtils Helper Object

`JdbcUtils` is a Scala object with [methods](#) to support `JDBCRDD`, `JDBCRelation` and `JdbcRelationProvider`.

Table 1. JdbcUtils API

Name	Description
createConnectionFactory	<p>Used when:</p> <ul style="list-style-type: none"> <code>JDBCRDD</code> is requested to <code>scanTable</code> and <code>resolveTable</code> <code>JdbcRelationProvider</code> is requested to <code>write the rows of a structured query (a DataFrame) to a table</code>
createTable	
dropTable	
getCommonJDBCType	
getCustomSchema	<p>Replaces data types in a table schema</p> <p>Used exclusively when <code>JDBCRelation</code> is <code>created</code> (and <code>customSchema</code> JDBC option was defined)</p>
getInsertStatement	
getSchema	Used when <code>JDBCRDD</code> is requested to <code>resolveTable</code>
getSchemaOption	Used when <code>JdbcRelationProvider</code> is requested to <code>write the rows of a structured query (a DataFrame) to a table</code>
resultSetToRows	Used when...FIXME
resultSetToSparkInternalRows	Used when <code>JDBCRDD</code> is requested to <code>compute a partition</code>
schemaString	
saveTable	
tableExists	Used when <code>JdbcRelationProvider</code> is requested to <code>write the rows of a structured query (a DataFrame) to a table</code>
truncateTable	Used when...FIXME

createConnectionFactory Method

```
createConnectionFactory(options: JDBCOptions): () => Connection
```

createConnectionFactory ...FIXME

Note

`createConnectionFactory` is used when:

- `JDBCRDD` is requested to `scanTable` (and in turn creates a `JDBCRDD`) and `resolveTable`
- `JdbcRelationProvider` is requested to `create a BaseRelation`
- `JdbcUtils` is requested to `saveTable`

getCommonJDBCType Method

```
getCommonJDBCType(dt: DataType): Option[JdbcType]
```

getCommonJDBCType ...FIXME

Note

`getCommonJDBCType` is used when...FIXME

getCatalystType Internal Method

```
getCatalystType(  
    sqlType: Int,  
    precision: Int,  
    scale: Int,  
    signed: Boolean): DataType
```

getCatalystType ...FIXME

Note

`getCatalystType` is used when...FIXME

getSchemaOption Method

```
getSchemaOption(conn: Connection, options: JDBCOptions): Option[StructType]
```

getSchemaOption ...FIXME

Note

`getSchemaOption` is used when...FIXME

getSchema Method

```
getSchema(  
    resultSet: ResultSet,  
    dialect: JdbcDialect,  
    alwaysNullable: Boolean = false): StructType
```

getSchema ...FIXME

Note	getSchema is used when...FIXME
------	--------------------------------

resultSetToRows Method

```
resultSetToRows(resultSet: ResultSet, schema: StructType): Iterator[Row]
```

resultSetToRows ...FIXME

Note	resultSetToRows is used when...FIXME
------	--------------------------------------

resultSetToSparkInternalRows Method

```
resultSetToSparkInternalRows(  
    resultSet: ResultSet,  
    schema: StructType,  
    inputMetrics: InputMetrics): Iterator[InternalRow]
```

resultSetToSparkInternalRows ...FIXME

Note	resultSetToSparkInternalRows is used when...FIXME
------	---

schemaString Method

```
schemaString(  
    df: DataFrame,  
    url: String,  
    createTableColumnTypes: Option[String] = None): String
```

schemaString ...FIXME

Note	schemaString is used exclusively when <code>JdbcUtils</code> is requested to create a table .
------	---

parseUserSpecifiedCreateTableColumnTypes Internal Method

```
parseUserSpecifiedCreateTableColumnTypes(
  df: DataFrame,
  createTableColumnTypes: String): Map[String, String]
```

`parseUserSpecifiedCreateTableColumnTypes` ...FIXME

Note	<code>parseUserSpecifiedCreateTableColumnTypes</code> is used exclusively when <code>JdbcUtils</code> is requested to <code>schemaString</code> .
------	---

saveTable Method

```
saveTable(
  df: DataFrame,
  tableSchema: Option[StructType],
  isCaseSensitive: Boolean,
  options: JDBCOptions): Unit
```

`saveTable` takes the `url`, `table`, `batchSize`, `isolationLevel` options and `createConnectionFactory`.

`saveTable` `getInsertStatement`.

`saveTable` takes the `numPartitions` option and applies `coalesce` operator to the input `DataFrame` if the number of partitions of its `RDD` is less than the `numPartitions` option.

In the end, `saveTable` requests the possibly-repartitioned `DataFrame` for its `RDD` (it may have changed after the `coalesce` operator) and executes `savePartition` for every partition (using `RDD.foreachPartition`).

Note	<code>saveTable</code> is used exclusively when <code>JdbcRelationProvider</code> is requested to write the rows of a structured query (a <code>DataFrame</code>) to a table.
------	--

Replacing Data Types In Table Schema — getCustomSchema Method

```
getCustomSchema(
  tableSchema: StructType,
  customSchema: String,
  nameEquality: Resolver): StructType
```

`getCustomSchema` replaces the data type of the fields in the input `tableSchema` schema that are included in the input `customSchema` (if defined).

Internally, `getCustomSchema` branches off per the input `customSchema`.

If the input `customSchema` is undefined or empty, `getCustomSchema` simply returns the input `tableSchema` unchanged.

Otherwise, if the input `customSchema` is not empty, `getCustomSchema` requests `catalystSqlParser` to parse it (i.e. create a new `StructType` for the given `customSchema` canonical schema representation).

`getCustomSchema` then uses `SchemaUtils` to `checkColumnNameDuplication` (in the column names of the user-defined `customSchema` schema with the input `nameEquality`).

In the end, `getCustomSchema` replaces the data type of the fields in the input `tableSchema` that are included in the input `userSchema`.

Note	<code>getCustomSchema</code> is used exclusively when <code>JDBCRelation</code> is created (and <code>customSchema</code> JDBC option was defined).
------	---

dropTable Method

```
dropTable(conn: Connection, table: String): Unit
```

`dropTable` ...FIXME

Note	<code>dropTable</code> is used when...FIXME
------	---

Creating Table Using JDBC — `createTable` Method

```
createTable(
  conn: Connection,
  df: DataFrame,
  options: JDBCOptions): Unit
```

`createTable` builds the table schema (given the input `DataFrame` with the `url` and `createTableColumnTypes` options).

`createTable` uses the `table` and `createTableOptions` options.

In the end, `createTable` concatenates all the above texts into a `CREATE TABLE [table] ([strSchema]) [createTableOptions]` SQL DDL statement followed by executing it (using the input JDBC `Connection`).

Note

`createTable` is used exclusively when `JdbcRelationProvider` is requested to write the rows of a structured query (a DataFrame) to a table.

getInsertStatement Method

```
getInsertStatement(
  table: String,
  rddSchema: StructType,
  tableSchema: Option[StructType],
  isCaseSensitive: Boolean,
  dialect: JdbcDialect): String
```

getInsertStatement ...FIXME

Note

`getInsertStatement` is used when...FIXME

getJdbcType Internal Method

```
getJdbcType(dt: DataType, dialect: JdbcDialect): JdbcType
```

getJdbcType ...FIXME

Note

`getJdbcType` is used when...FIXME

tableExists Method

```
tableExists(conn: Connection, options: JDBCOptions): Boolean
```

tableExists ...FIXME

Note

`tableExists` is used exclusively when `JdbcRelationProvider` is requested to write the rows of a structured query (a DataFrame) to a table.

truncateTable Method

```
truncateTable(conn: Connection, options: JDBCOptions): Unit
```

truncateTable ...FIXME

Note

`truncateTable` is used exclusively when `JdbcRelationProvider` is requested to write the rows of a structured query (a DataFrame) to a table.

Saving Rows (Per Partition) to Table — `savePartition` Method

```
savePartition(  
    getConnection: () => Connection,  
    table: String,  
    iterator: Iterator[Row],  
    rddSchema: StructType,  
    insertStmt: String,  
    batchSize: Int,  
    dialect: JdbcDialect,  
    isolationLevel: Int): Iterator[Byte]
```

`savePartition` creates a JDBC `Connection` using the input `getConnection` function.

`savePartition` tries to set the input `isolationLevel` if it is different than `TRANSACTION_NONE` and the database supports transactions.

`savePartition` then writes rows (in the input `Iterator[Row]`) using batches that are submitted after `batchSize` rows where added.

Note

`savePartition` is used exclusively when `JdbcUtils` is requested to `saveTable`.

Console Data Source

Console Data Source is...FIXME

ConsoleSinkProvider

ConsoleSinkProvider is...FIXME

Hive Integration

Spark SQL supports [Apache Hive](#) using `HiveContext`. It uses the Spark SQL execution engine to work with data stored in Hive.

From [Wikipedia, the free encyclopedia](#):

Apache Hive supports analysis of large datasets stored in Hadoop's HDFS and compatible file systems such as Amazon S3 filesystem.

Note

It provides an SQL-like language called HiveQL with schema on read and transparently converts queries to Hadoop MapReduce, Apache Tez and Apache Spark jobs.

All three execution engines can run in Hadoop YARN.

`HiveContext` is a specialized `SQLContext` to work with Hive.

There is also a dedicated tool [spark-sql](#) that...FIXME

Tip

Import `org.apache.spark.sql.hive` package to use `HiveContext`.

Enable `DEBUG` logging level for `HiveContext` to see what happens inside.

Add the following line to `conf/log4j.properties`:

Tip

```
log4j.logger.org.apache.spark.sql.hive.HiveContext=DEBUG
```

Refer to [Logging](#).

Hive Functions

[SQLContext.sql](#) (or simply `sql`) allows you to interact with Hive.

You can use `show functions` to learn about the Hive functions supported through the Hive integration.

```

scala> sql("show functions").show(false)
16/04/10 15:22:08 INFO HiveSqlParser: Parsing command: show functions
+-----+
|function      |
+-----+
| !
|%
|&
|*
|+
|-|
|/
|<
|<=
|<=>
|=|
|==|
|>
|>=
|^
|abs
|acos
|add_months
|and
|approx_count_distinct|
+-----+
only showing top 20 rows

```

Hive Configuration - `hive-site.xml`

The configuration for Hive is in `hive-site.xml` on the classpath.

The default configuration uses Hive 1.2.1 with the default warehouse in `/user/hive/warehouse`.

```

16/04/09 13:37:54 INFO HiveContext: Initializing execution hive, version 1.2.1
16/04/09 13:37:58 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so recording the schema version 1.2.0
16/04/09 13:37:58 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
16/04/09 13:37:58 INFO HiveContext: default warehouse location is /user/hive/warehouse
16/04/09 13:37:58 INFO HiveContext: Initializing HiveMetastoreConnection version 1.2.1 using Spark classes.
16/04/09 13:38:01 DEBUG HiveContext: create HiveContext

```

current_database function

`current_database` function returns the current database of Hive metadata.

```
scala> sql("select current_database()").show(false)
16/04/09 13:52:13 INFO HiveSqlParser: Parsing command: select current_database()
+-----+
|currentdatabase()|
+-----+
|default      |
+-----+
```

`current_database` function is registered when `HiveContext` is initialized.

Internally, it uses private `CurrentDatabase` class that uses
`HiveContext.sessionState.catalog.getCurrentDatabase`.

Analyzing Tables

```
analyze(tableName: String)
```

`analyze` analyzes `tableName` table for query optimizations. It currently supports only Hive tables.

```
scala> sql("show tables").show(false)
16/04/09 14:04:10 INFO HiveSqlParser: Parsing command: show tables
+-----+
|tableName|isTemporary|
+-----+
|dafa     |false      |
+-----+
```

```
scala> spark.asInstanceOf[HiveContext].analyze("dafa")
16/04/09 14:02:56 INFO HiveSqlParser: Parsing command: dafa
java.lang.UnsupportedOperationException: Analyze only works for Hive tables, but dafa
is a LogicalRelation
    at org.apache.spark.sql.hive.HiveContext.analyze(HiveContext.scala:304)
    ... 50 elided
```

Experimental: Metastore Tables with non-Hive SerDe

Caution

FIXME Review the uses of `convertMetastoreParquet`,
`convertMetastoreParquetWithSchemaMerging`, `convertMetastoreOrc`,
`convertCTAS`.

Hive Metastore

Spark SQL uses a Hive metastore to manage the metadata of persistent relational entities (e.g. databases, tables, columns, partitions) in a relational database (for fast access).

A Hive metastore warehouse (aka [spark-warehouse](#)) is the directory where Spark SQL persists tables whereas a Hive metastore (aka [metastore_db](#)) is a relational database to manage the metadata of the persistent relational entities, e.g. databases, tables, columns, partitions.

By default, Spark SQL uses the embedded deployment mode of a Hive metastore with a [Apache Derby](#) database.

Important	The default embedded deployment mode is not recommended for production use due to limitation of only one active SparkSession at a time. Read Cloudera's Configuring the Hive Metastore for CDH document that explains the available deployment modes of a Hive metastore.
-----------	--

When `SparkSession` is [created with Hive support](#) the external catalog (aka *metastore*) is `HiveExternalCatalog`. `HiveExternalCatalog` uses `spark.sql.warehouse.dir` directory for the location of the databases and `javax.jdo.option properties` for the connection to the Hive metastore database.

Note	The metadata of relational entities is persisted in a metastore database over JDBC and DataNucleus AccessPlatform that uses <code>javax.jdo.option</code> properties. Read Hive Metastore Administration to learn how to manage a Hive Metastore.
------	--

Table 1. Hive Metastore Database Connection Properties

Name	Description
javax.jdo.option.ConnectionURL	The JDBC connection URL of a Hive metastore jdbc:derby:;databaseName=metastore_db;create=true jdbc:derby:memory:;databaseName=\${metastoreL... jdbc:mysql://192.168.175.160:3306/metastore?...
javax.jdo.option.ConnectionDriverName	The JDBC driver of a Hive metastore database org.apache.derby.jdbc.EmbeddedDriver
javax.jdo.option.ConnectionUserName	The user name to use to connect to a Hive metastore
javax.jdo.option.ConnectionPassword	The password to use to connect to a Hive metastore

You can configure `javax.jdo.option` properties in [hive-site.xml](#) or using options with `spark.hadoop` prefix.

You can access the current connection properties for a Hive metastore in a Spark SQL application using the Spark internal classes.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> spark.sharedState.externalCatalog
res1: org.apache.spark.sql.catalyst.catalog.ExternalCatalog = org.apache.spark.sql.hive.HiveExternalCatalog@79dd79eb

// Use `:paste -raw` to paste the following code
// This is to pass the private[spark] "gate"
// BEGIN
package org.apache.spark
import org.apache.spark.sql.SparkSession
object jacek {
  def open(spark: SparkSession) = {
    import org.apache.spark.sql.hive.HiveExternalCatalog
    spark.sharedState.externalCatalog.asInstanceOf[HiveExternalCatalog].client
  }
}
// END
import org.apache.spark.jacek
val hiveClient = jacek.open(spark)
scala> hiveClient.getConf("javax.jdo.option.ConnectionURL", "")
res2: String = jdbc:derby:;databaseName=metastore_db;create=true
```

The benefits of using an external Hive metastore:

1. Allow multiple Spark applications (sessions) to access it concurrently
2. Allow a single Spark application to use table statistics without running "ANALYZE TABLE" every execution

Note	As of Spark 2.2 (see SPARK-18112 <code>Spark2.x</code> does not support read data from <code>Hive 2.x metastore</code>) Spark SQL supports reading data from Hive 2.1.1 metastore.
------	---

Caution	FIXME Describe <code>hive-site.xml</code> vs <code>config</code> method vs <code>--conf</code> with <code>spark.hadoop</code> prefix.
---------	---

Spark SQL uses the Hive-specific configuration properties that further fine-tune the Hive integration, e.g. `spark.sql.hive.metastore.version` or `spark.sql.hive.metastore.jars`.

spark.sql.warehouse.dir Configuration Property

`spark.sql.warehouse.dir` is a static configuration property that sets Hive's `hive.metastore.warehouse.dir` property, i.e. the location of the Hive local/embedded metastore database (using Derby).

Tip	Refer to SharedState to learn about (the low-level details of) Spark SQL support for Apache Hive. See also the official Hive Metastore Administration document.
-----	--

Hive Metastore Deployment Modes

Configuring External Hive Metastore in Spark SQL

In order to use an external Hive metastore you should do the following:

1. Enable Hive support in `SparkSession` (that makes sure that the Hive classes are on CLASSPATH and sets `spark.sql.catalogImplementation` internal configuration property to `hive`)
2. `spark.sql.warehouse.dir` required?
3. Define `hive.metastore.warehouse.dir` in `hive-site.xml` configuration resource
4. Check out `warehousePath`
5. Execute `./bin/run-example sql.hive.SparkHiveExample` to verify Hive configuration

When not configured by the `hive-site.xml`, `SparkSession` automatically creates `metastore_db` in the current directory and creates a directory configured by `spark.sql.warehouse.dir`, which defaults to the directory `spark-warehouse` in the current directory that the Spark application is started.

Note <p><code>hive.metastore.warehouse.dir</code> property in <code>hive-site.xml</code> is deprecated since Spark 2.0.0. Use <code>spark.sql.warehouse.dir</code> to specify the default location of the databases in a Hive warehouse.</p> <p>You may need to grant write privilege to the user who starts the Spark application.</p>

Hadoop Configuration Properties for Hive

Table 2. Hadoop Configuration Properties for Hive

Name	Description
<code>hive.metastore.uris</code>	The Thrift URI of a remote Hive metastore, i.e. one separate JVM process or on a remote node config("hive.metastore.uris", "thrift://192.168.1.10:10000")
<code>hive.metastore.warehouse.dir</code>	SharedState uses <code>hive.metastore.warehouse.dir</code> to <code>spark.sql.warehouse.dir</code> if the latter is undefined. Caution FIXME How is <code>hive.metastore.warehouse.dir</code> related to <code>spark.sql.warehouse.dir</code> ? SharedState.warehousePath? Review https://github.com/apache/spark/pull/10000
<code>hive.metastore.schema.verification</code>	Set to <code>false</code> (as seems to cause exceptions with metastore database as of Hive 2.1)

You may also want to use the following Hive configuration properties that (seem to) cause exceptions with an empty metastore database as of Hive 2.1.

- `datanucleus.schema.autoCreateAll` set to `true`

spark.hadoop Configuration Properties

Caution <p>FIXME Describe the purpose of <code>spark.hadoop.*</code> properties</p>

You can specify any of the Hadoop configuration properties, e.g. `hive.metastore.warehouse.dir` with **spark.hadoop** prefix.

```
$ spark-shell --conf spark.hadoop.hive.metastore.warehouse.dir=/tmp/hive-warehouse
...
scala> spark.sharedState
18/01/08 10:46:19 INFO SharedState: spark.sql.warehouse.dir is not set, but hive.metastore.warehouse.dir is set. Setting spark.sql.warehouse.dir to the value of hive.metastore.warehouse.dir ('/tmp/hive-warehouse').
18/01/08 10:46:19 INFO SharedState: Warehouse path is '/tmp/hive-warehouse'.
res1: org.apache.spark.sql.internal.SharedState = org.apache.spark.sql.internal.SharedState@5a69b3cf
```

hive-site.xml Configuration Resource

`hive-site.xml` configures Hive clients (e.g. Spark SQL) with the Hive Metastore configuration.

`hive-site.xml` is loaded when [SharedState](#) is created (which is...FIXME).

Configuration of Hive is done by placing your `hive-site.xml`, `core-site.xml` (for security configuration), and `hdfs-site.xml` (for HDFS configuration) file in `conf/` (that is automatically added to the CLASSPATH of a Spark application).

Tip	You can use <code>--driver-class-path</code> or <code>spark.driver.extraClassPath</code> to point to the directory with configuration resources, e.g. <code>hive-site.xml</code> .
-----	--

```
<configuration>
<property>
  <name>hive.metastore.warehouse.dir</name>
  <value>/tmp/hive-warehouse</value>
  <description>Hive Metastore location</description>
</property>
</configuration>
```

Tip	Read Resources section in Hadoop's Configuration javadoc to learn more about configuration resources.
-----	--

Tip Use `SparkContext.hadoopConfiguration` to know which configuration resources have

```
scala> sc.hadoopConfiguration
res1: org.apache.hadoop.conf.Configuration = Configuration: core-default.xml, co
// Initialize warehousePath
scala> spark.sharedState.warehousePath
res2: String = file:/Users/jacek/dev/oss/spark/spark-warehouse/
// Note file:/Users/jacek/dev/oss/spark/spark-warehouse/ is added to configuration
scala> sc.hadoopConfiguration
res3: org.apache.hadoop.conf.Configuration = Configuration: core-default.xml, co
```

Tip

Enable `org.apache.spark.sql.internal.SharedState` logger to `INFO` logging level to

```
scala> spark.sharedState.warehousePath
18/01/08 09:49:33 INFO SharedState: loading hive config file: file:/Users/jacek/
18/01/08 09:49:33 INFO SharedState: Setting hive.metastore.warehouse.dir ('null')
18/01/08 09:49:33 INFO SharedState: Warehouse path is 'file:/Users/jacek/dev/oss'
res2: String = file:/Users/jacek/dev/oss/spark/spark-warehouse/
```

Starting Hive

The following steps are for Hive and Hadoop 2.7.5.

```
$ ./bin/hdfs version
Hadoop 2.7.5
Subversion https://shv@git-wip-us.apache.org/repos/asf/hadoop.git -r 18065c2b6806ed4aa
6a3187d77cbe21bb3dba075
Compiled by kshvachk on 2017-12-16T01:06Z
Compiled with protoc 2.5.0
From source with checksum 9f118f95f47043332d51891e37f736e9
This command was run using /Users/jacek/dev/apps/hadoop-2.7.5/share/hadoop/common/hado
op-common-2.7.5.jar
```

Tip

Read the section [Pseudo-Distributed Operation](#) about how to run Hadoop HDFS *"on a single-node in a pseudo-distributed mode where each Hadoop daemon runs in a separate Java process."*

Use `hadoop.tmp.dir` configuration property as the base for temporary directories.

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>/tmp/my-hadoop-tmp-dir/hdfs/tmp</value>
  <description>The base for temporary directories.</description>
</property>
```

Tip

Use `./bin/hdfs getconf -confKey hadoop.tmp.dir` to check out the value

```
$ ./bin/hdfs getconf -confKey hadoop.tmp.dir
/tmp/my-hadoop-tmp-dir/hdfs/tmp
```

1. Edit `etc/hadoop/core-site.xml` to add the following:

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

2. `./bin/hdfs namenode -format` right after you've installed Hadoop and before starting any HDFS services (NameNode in particular)

```
$ ./bin/hdfs namenode -format
18/01/09 15:48:28 INFO namenode.NameNode: STARTUP_MSG:
*****
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = japila.local/192.168.1.2
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 2.7.5
...
18/01/09 15:48:28 INFO namenode.NameNode: createNameNode [-format]
...
Formatting using clusterid: CID-bfdc81da-6941-4a93-8371-2c254d503a97
...
18/01/09 15:48:29 INFO common.Storage: Storage directory /tmp/hadoop-jacek/dfs/nam
e has been successfully formatted.
18/01/09 15:48:29 INFO namenode.FSImageFormatProtobuf: Saving image file /tmp/hado
op-jacek/dfs/name/current/fsimage.ckpt_00000000000000000000 using no compression
18/01/09 15:48:29 INFO namenode.FSImageFormatProtobuf: Image file /tmp/hadoop-jace
k/dfs/name/current/fsimage.ckpt_00000000000000000000 of size 322 bytes saved in 0 s
econds.
18/01/09 15:48:29 INFO namenode.NNStorageRetentionManager: Going to retain 1 image
s with txid >= 0
18/01/09 15:48:29 INFO util.ExitUtil: Exiting with status 0
```

Use `./bin/hdfs namenode` to start a NameNode that will tell you that the local

Note

```
$ ./bin/hdfs namenode
18/01/09 15:43:11 INFO namenode.NameNode: STARTUP_MSG:
/*****
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = japila.local/192.168.1.2
STARTUP_MSG:   args = []
STARTUP_MSG:   version = 2.7.5
...
18/01/09 15:43:11 INFO namenode.NameNode: fs.defaultFS is hdfs://localhost
18/01/09 15:43:11 INFO namenode.NameNode: Clients are to use localhost:900
...
18/01/09 15:43:12 INFO hdfs.DFSUtil: Starting Web-server for hdfs at: http
...
18/01/09 15:43:13 WARN common.Storage: Storage directory /private/tmp/hado
18/01/09 15:43:13 WARN namenode.FSNamesystem: Encountered exception loadin
org.apache.hadoop.hdfs.server.common.InconsistentFSStateException: Directo
    at org.apache.hadoop.hdfs.server.namenode.FSImage.recoverStorageDi
    at org.apache.hadoop.hdfs.server.namenode.FSImage.recoverTransitio
    at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.loadFSImage
    at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.loadFromDis
    at org.apache.hadoop.hdfs.server.namenode.NameNode.loadNamesystem(
    at org.apache.hadoop.hdfs.server.namenode.NameNode.initialize(Name
    at org.apache.hadoop.hdfs.server.namenode.NameNode.<init>(NameNode
    at org.apache.hadoop.hdfs.server.namenode.NameNode.<init>(NameNode
    at org.apache.hadoop.hdfs.server.namenode.NameNode.createNameNode(
    at org.apache.hadoop.hdfs.server.namenode.NameNode.main(NameNode.j
...
18/01/09 15:43:13 ERROR namenode.NameNode: Failed to start namenode.
org.apache.hadoop.hdfs.server.common.InconsistentFSStateException: Directo
    at org.apache.hadoop.hdfs.server.namenode.FSImage.recoverStorageDi
    at org.apache.hadoop.hdfs.server.namenode.FSImage.recoverTransitio
    at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.loadFSImage
    at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.loadFromDis
    at org.apache.hadoop.hdfs.server.namenode.NameNode.loadNamesystem(
    at org.apache.hadoop.hdfs.server.namenode.NameNode.initialize(Name
    at org.apache.hadoop.hdfs.server.namenode.NameNode.<init>(NameNode
    at org.apache.hadoop.hdfs.server.namenode.NameNode.<init>(NameNode
    at org.apache.hadoop.hdfs.server.namenode.NameNode.createNameNode(
    at org.apache.hadoop.hdfs.server.namenode.NameNode.main(NameNode.j
```

- Start Hadoop HDFS using `./sbin/start-dfs.sh` (and `tail -f logs/hadoop-* -datanode-* .log`)

```
$ ./sbin/start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /Users/jacek/dev/apps/hadoop-2.7.5/logs/h
adoop-jacek-namenode-japila.local.out
localhost: starting datanode, logging to /Users/jacek/dev/apps/hadoop-2.7.5/logs/h
adoop-jacek-datanode-japila.local.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /Users/jacek/dev/apps/hadoop-2.7.5
/logs/hadoop-jacek-secondarynamenode-japila.local.out
```

- Use `jps -lm` to list Hadoop's JVM processes.

```
$ jps -lm
26576 org.apache.hadoop.hdfs.server.namenode.SecondaryNameNode
26468 org.apache.hadoop.hdfs.server.datanode.DataNode
26381 org.apache.hadoop.hdfs.server.namenode.NameNode
```

5. Create `hive-site.xml` in `$SPARK_HOME/conf` with the following:

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>hdfs://localhost:9000/jacek/hive_warehouse</value>
    <description>Warehouse Location</description>
  </property>
</configuration>
```

Spark SQL CLI — spark-sql

Caution	FIXME
---------	-------

Tip	Read about Spark SQL CLI in Spark's official documentation in Running the Spark SQL CLI .
-----	---

```
spark-sql> describe function `<>`;  
Function: <>  
Usage: a <> b - Returns TRUE if a is not equal to b
```

Tip	Functions are registered in FunctionRegistry .
-----	--

```
spark-sql> show functions;
```

```
spark-sql> explain extended show tables;
```

DataSinks

Caution	FIXME
---------	-------

HiveFileFormat — FileFormat For Writing Hive Tables

`HiveFileFormat` is a [FileFormat](#) for writing Hive tables.

`HiveFileFormat` is a [DataSourceRegister](#) and registers itself as `hive` data source.

Note

Hive data source can only be used with tables and you cannot read or write files of Hive data source directly. Use [DataFrameReader.table](#) or [DataFrameWriter.saveAsTable](#) for loading from or writing data to Hive data source, respectively.

`HiveFileFormat` is [created](#) exclusively when `SaveAsHiveFile` is requested to [saveAsHiveFile](#) (when [InsertIntoHiveDirCommand](#) and [InsertIntoHiveTable](#) logical commands are executed).

`HiveFileFormat` takes a `FileSinkDesc` when created.

`HiveFileFormat` throws a `UnsupportedOperationException` when requested to [inferSchema](#).

`inferSchema` is not supported for hive data source.

Preparing Write Job — `prepareWrite` Method

```
prepareWrite(
    sparkSession: SparkSession,
    job: Job,
    options: Map[String, String],
    dataSchema: StructType): OutputWriterFactory
```

Note

`preparewrite` is part of the [FileFormat Contract](#) to prepare a write job.

`prepareWrite` sets the `mapred.output.format.class` property to be the `getOutputFormatClassName` of the Hive `TableDesc` of the [FileSinkDesc](#).

`prepareWrite` requests the `HiveTableUtil` helper object to `configureJobPropertiesForStorageHandler`.

`prepareWrite` requests the Hive `Utilities` helper object to `copyTableJobPropertiesToConf`.

In the end, `preparewrite` creates a new `OutputWriterFactory` that creates a new `HiveOutputWriter` when requested for a new `OutputWriter` instance.

HiveClient — Contract for Retrieving Metadata from Hive Metastore

`HiveClient` is the [contract](#) for...FIXME

Note	<code>HiveClientImpl</code> is the only available <code>HiveClient</code> in Spark SQL.
------	---

`HiveClient` offers [safe](#) variants of many methods that do not report exceptions when a relational entity is not found in a Hive metastore, e.g. [getTableOption](#) for [getTable](#).

```
package org.apache.spark.sql.hive.client

trait HiveClient {
    // only required methods that have no implementation
    // FIXME List of the methods
    def alterPartitions(
        db: String,
        table: String,
        newParts: Seq[CatalogTablePartition]): Unit
    def getTableOption(dbName: String, tableName: String): Option[CatalogTable]
    def getPartitions(
        catalogTable: CatalogTable,
        partialSpec: Option[TablePartitionSpec] = None): Seq[CatalogTablePartition]
    def getPartitionsByFilter(
        catalogTable: CatalogTable,
        predicates: Seq[Expression]): Seq[CatalogTablePartition]
    def getPartitionOption(
        table: CatalogTable,
        spec: TablePartitionSpec): Option[CatalogTablePartition]
    def renamePartitions(
        db: String,
        table: String,
        specs: Seq[TablePartitionSpec],
        newSpecs: Seq[TablePartitionSpec]): Unit
}
```

Note	<code>HiveClient</code> is a <code>private[hive]</code> contract.
------	---

Table 1. (Subset of) HiveClient Contract

Method	Description		
alterPartitions			
getPartitions	<pre>getPartitions(db: String, table: String, partialSpec: Option[TablePartitionSpec]): Seq[CatalogTablePartition] getPartitions(catalogTable: CatalogTable, partialSpec: Option[TablePartitionSpec] = None): Seq[CatalogTablePartition]</pre> <p>Returns the CatalogTablePartition of a table Used exclusively when <code>HiveExternalCatalog</code> is requested to list partitions of a table.</p>		
getPartitionsByFilter	Used when...FIXME		
getPartitionOption	Used when...FIXME		
getTableOption	<pre>getTableOption(dbName: String, tableName: String): Option[CatalogTableOption]</pre> <p>Retrieves a table metadata if available Used exclusively when <code>HiveClient</code> is requested for a table metadata</p> <table border="1" style="margin-left: 20px;"> <tr> <td>Note</td> <td><code>getTableOption</code> is a safe version of getTable as it does not throw a <code>NoSuchTableException</code>, but simply returns <code>None</code> if no such table exists.</td> </tr> </table>	Note	<code>getTableOption</code> is a safe version of getTable as it does not throw a <code>NoSuchTableException</code> , but simply returns <code>None</code> if no such table exists.
Note	<code>getTableOption</code> is a safe version of getTable as it does not throw a <code>NoSuchTableException</code> , but simply returns <code>None</code> if no such table exists.		
renamePartitions	Used when...FIXME		

Retrieving Table Metadata If Available or Throwing `NoSuchTableException` — `getTable` Method

```
getTable(dbName: String, tableName: String): CatalogTable
```

`getTable` retrieves the metadata of a table in a Hive metastore if available or reports a `NoSuchTableException`.

	<p><code>getTable</code> is used when:</p> <ul style="list-style-type: none">• <code>HiveExternalCatalog</code> is requested for a table metadata• <code>HiveClient</code> is requested for getPartitionOption or getPartitions• <code>HiveClientImpl</code> is requested for renamePartitions or alterPartitions
--	---

HiveClientImpl — The One and Only HiveClient

`HiveClientImpl` is the only available [HiveClient](#) in Spark SQL that does/uses...FIXME

`HiveClientImpl` is [created](#) exclusively when `IsolatedClientLoader` is requested to [create a new Hive client](#). When created, `HiveClientImpl` is given the location of the default database for the Hive metastore warehouse (i.e. `warehouseDir` that is the value of `hive.metastore.warehouse.dir` Hive-specific Hadoop configuration property).

Note	The location of the default database for the Hive metastore warehouse is <code>/user/hive/warehouse</code> by default.
Note	You may be interested in SPARK-19664 put 'hive.metastore.warehouse.dir' in hadoopConf place if you use Spark before 2.1 (which you should not really as it is not supported anymore).
Note	The Hadoop configuration is what HiveExternalCatalog was given when created (which is the default Hadoop configuration from Spark Core's <code>SparkContext.hadoopConfiguration</code> with the Spark properties with <code>spark.hadoop</code> prefix).
Tip	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.sql.hive.client.HiveClientImpl</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.hive.client.HiveClientImpl=DEBUG</pre> <p>Refer to Logging.</p>

renamePartitions Method

```
renamePartitions(
  db: String,
  table: String,
  specs: Seq[TablePartitionSpec],
  newSpecs: Seq[TablePartitionSpec]): Unit
```

Note	<code>renamePartitions</code> is part of HiveClient Contract to...FIXME.
------	--

`renamePartitions` ...FIXME

alterPartitions Method

```
alterPartitions(  
    db: String,  
    table: String,  
    newParts: Seq[CatalogTablePartition]): Unit
```

Note

`alterPartitions` is part of [HiveClient Contract](#) to...FIXME.

`alterPartitions` ...FIXME

client Internal Method

```
client: Hive
```

`client` ...FIXME

Note

`client` is used...FIXME

getPartitions Method

```
getPartitions(  
    table: CatalogTable,  
    spec: Option[TablePartitionSpec]): Seq[CatalogTablePartition]
```

Note

`getPartitions` is part of [HiveClient Contract](#) to...FIXME.

`getPartitions` ...FIXME

getPartitionsByFilter Method

```
getPartitionsByFilter(  
    table: CatalogTable,  
    predicates: Seq[Expression]): Seq[CatalogTablePartition]
```

Note

`getPartitionsByFilter` is part of [HiveClient Contract](#) to...FIXME.

`getPartitionsByFilter` ...FIXME

getPartitionOption Method

```
getPartitionOption(  
    table: CatalogTable,  
    spec: TablePartitionSpec): Option[CatalogTablePartition]
```

Note

`getPartitionOption` is part of [HiveClient Contract](#) to...FIXME.

`getPartitionOption` ...FIXME

Creating HiveClientImpl Instance

`HiveClientImpl` takes the following when created:

- `HiveVersion`
- Location of the default database for the Hive metastore warehouse if defined (aka `warehouseDir`)
- `SparkConf`
- Hadoop configuration
- Extra configuration
- Initial `ClassLoader`
- `IsolatedClientLoader`

`HiveClientImpl` initializes the [internal registries and counters](#).

Retrieving Table Metadata If Available — `getTableOption` Method

```
def getTableOption(dbName: String, tableName: String): Option[CatalogTable]
```

Note

`getTableOption` is part of [HiveClient Contract](#) to...FIXME.

When executed, `getTableOption` prints out the following DEBUG message to the logs:

```
Looking up [dbName].[tableName]
```

`getTableOption` requests [Hive client](#) to retrieve the metadata of the table and creates a [CatalogTable](#).

Creating Table Statistics from Hive's Table or Partition Parameters — `readHiveStats` Internal Method

```
readHiveStats(properties: Map[String, String]): Option[CatalogStatistics]
```

`readHiveStats` creates a [CatalogStatistics](#) from the input Hive table or partition parameters (if available and greater than 0).

Table 1. Table Statistics and Hive Parameters

Hive Parameter	Table Statistics
<code>totalSize</code>	<code>sizeInBytes</code>
<code>rawDataSize</code>	<code>sizeInBytes</code>
<code>numRows</code>	<code>rowCount</code>

Note `totalSize` Hive parameter has a higher precedence over `rawDataSize` for `sizeInBytes` table statistic.

Note `readHiveStats` is used when `HiveClientImpl` is requested for the metadata of a [table](#) or [table partition](#).

Retrieving Table Partition Metadata (Converting Table Partition Metadata from Hive Format to Spark SQL Format) — `fromHivePartition` Method

```
fromHivePartition(hp: HivePartition): CatalogTablePartition
```

`fromHivePartition` simply creates a [CatalogTablePartition](#) with the following:

- `spec` from Hive's `Partition.getSpec` if available
- `storage` from Hive's `StorageDescriptor` of the table partition
- `parameters` from Hive's `Partition.getParameters` if available
- `stats` from Hive's `Partition.getParameters` if available and converted to table statistics format

Note `fromHivePartition` is used when `HiveClientImpl` is requested for `getPartitionOption`, `getPartitions` and `getPartitionsByFilter`.

Converting Native Table Metadata to Hive's Table — `toHiveTable` Method

```
toHiveTable(table: CatalogTable, userName: Option[String] = None): HiveTable
```

`toHiveTable` simply creates a new Hive `Table` and copies the properties from the input `CatalogTable`.

Note

`toHiveTable` is used when:

- `HiveUtils` is requested to `inferSchema`
- `HiveClientImpl` is requested to `createTable`, `alterTable`, `renamePartitions`, `alterPartitions`, `getPartitionOption`, `getPartitions` and `getPartitionsByFilter`
- `HiveTableScanExec` physical operator is requested for the `hiveQlTable`
- `InsertIntoHiveDirCommand` and `InsertIntoHiveTable` logical commands are executed

getSparkSQLDataType Internal Utility

```
getSparkSQLDataType(hc: FieldSchema): DataType
```

`getSparkSQLDataType` ...FIXME

Note

`getSparkSQLDataType` is used when...FIXME

HiveUtils

`HiveUtils` is used to create a `HiveClientImpl` that `HiveExternalCatalog` uses to interact with a Hive metastore.

Tip

Enable `INFO` logging level for `org.apache.spark.sql.hive.HiveUtils` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.hive.HiveUtils=INFO
```

Refer to [Logging](#).

Creating `HiveClientImpl`— `newClientForMetadata` Method

```
newClientForMetadata(  
    conf: SparkConf,  
    hadoopConf: Configuration): HiveClient (1)  
newClientForMetadata(  
    conf: SparkConf,  
    hadoopConf: Configuration,  
    configurations: Map[String, String]): HiveClient
```

1. Executes the other `newClientForMetadata` with time configurations formatted

Internally, `newClientForMetadata` creates a new `SQLConf` with `spark.sql` properties only (from the input `SparkConf`).

`newClientForMetadata` then creates a `IsolatedClientLoader` per the input parameters and the following configuration properties:

- `spark.sql.hive.metastore.version`
- `spark.sql.hive.metastore.jars`
- `spark.sql.hive.metastore.sharedPrefixes`
- `spark.sql.hive.metastore.barrierPrefixes`

You should see one of the following INFO messages in the logs:

```
Initializing HiveMetastoreConnection version [hiveMetastoreVersion] using Spark classe  
s.  
Initializing HiveMetastoreConnection version [hiveMetastoreVersion] using maven.  
Initializing HiveMetastoreConnection version [hiveMetastoreVersion] using [jars]
```

In the end, `newClientForMetadata` requests the `IsolatedClientLoader` to create a [HiveClientImpl](#).

Note

`newClientForMetadata` is used exclusively when `HiveExternalCatalog` is requested for a [HiveClient](#).

inferSchema Method

```
inferSchema(table: CatalogTable): CatalogTable
```

`inferSchema` ...FIXME

Note

`inferSchema` is used when...FIXME

DataSourceV2 — Data Sources in Data Source API V2

`DataSourceV2` is the fundamental abstraction of the [data sources](#) in the [Data Source API V2](#).

`DataSourceV2` defines no methods or values and simply acts as a **marker interface**.

```
package org.apache.spark.sql.sources.v2;

public interface DataSourceV2 {}
```

	<p>Implementations should at least use ReadSupport or WriteSupport interfaces for reading and writing data sources, respectively.</p> <p>Otherwise, an <code>AnalysisException</code> is thrown:</p>
Note	<pre>org.apache.spark.sql.AnalysisException: dawid is not a valid Spark SQL Data Source at org.apache.spark.sql.execution.datasources.DataSource.resolveRelation(DataSour... at org.apache.spark.sql.DataFrameReader.loadV1Source(DataFrameReader.scala:222) at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:208) at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:167) 49 elided</pre>

`DataSourceV2` is an [Evolving](#) contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.

In other words, using the contract is as treading on thin ice.

Table 1. DataSourceV2s

DataSourceV2	Description
ConsoleSinkProvider	Used in Spark Structured Streaming
ContinuousReadSupport	Used in Spark Structured Streaming
MemorySinkV2	Used in Spark Structured Streaming
MicroBatchReadSupport	Used in Spark Structured Streaming
RateSourceProvider	Used in Spark Structured Streaming
RateSourceProviderV2	Used in Spark Structured Streaming
ReadSupport	
ReadSupportWithSchema	
SessionConfigSupport	
StreamWriteSupport	
WriteSupport	

ReadSupport Contract — "Readable" Data Sources

`ReadSupport` is the abstraction of "readable" data sources in the [Data Source API V2](#) that can create a [DataSourceReader](#) for reading data (*data scan*).

`ReadSupport` defines a single `createReader` method that creates a [DataSourceReader](#).

```
DataSourceReader createReader(DataSourceOptions options)
DataSourceReader createReader(StructType schema, DataSourceOptions options)
```

`createReader` is used when `DataSourceV2Relation` leaf logical operator is [created](#) (when `DataFrameReader` is requested to "load" data (as a `DataFrame`) from a data source with `ReadSupport`).

```
// FIXME: Demo
// spark.read.format(...) that is DataSourceV2 and ReadSupport
// DataFrameReader.load() creates a DataFrame with a DataSourceV2Relation operator
```

Internally, `ReadSupport` is accessed implicitly when `DataSourceV2Relation` logical operator is requested to [create a `DataSourceReader`](#).

Note	There are no production implementations of the ReadSupport Contract in Spark SQL yet.
------	---

WriteSupport Contract — "Writable" Data Sources

`writeSupport` is the abstraction of "writable" data sources in the [Data Source API V2](#) that can create a [DataSourceWriter](#) for writing data out.

`WriteSupport` defines a single `createWriter` method that creates an optional [DataSourceWriter](#) per [SaveMode](#) (and can create no `DataSourceWriter` when not needed per mode)

```
Optional<DataSourceWriter> createWriter(  
    String writeUUID,  
    StructType schema,  
    SaveMode mode,  
    DataSourceOptions options)
```

`createWriter` is used when:

- `DataFrameWriter` is requested to save a `DataFrame` to a data source (for [DataSourceV2](#) data sources with [WriteSupport](#))
- `DataSourceV2Relation` leaf logical operator is requested to [create a DataSourceWriter](#) (indirectly via `createWriter` implicit method)

```
// FIXME: Demo  
// df.write.format(...) that is DataSourceV2 and WriteSupport
```

Note

There are no production implementations of the [WriteSupport Contract](#) in Spark SQL yet.

DataSourceReader Contract

`DataSourceReader` is the [abstraction](#) of [data source readers](#) in Data Source API V2 that can [plan InputPartitions](#) and know the [schema](#) for reading.

`DataSourceReader` is created to scan the data from a data source when:

- `DataSourceV2Relation` is requested to [create a new reader](#)
- `ReadSupport` is requested to [create a reader](#)

`DataSourceReader` is used to create `StreamingDataSourceV2Relation` and [DataSourceV2ScanExec](#) physical operator

Note	It <i>appears</i> that all concrete data source readers are used in Spark Structured Streaming only.
------	--

Table 1. DataSourceReader Contract

Method	Description
<code>planInputPartitions</code>	<pre data-bbox="589 287 1356 316"><code>List<InputPartition<InternalRow>> planInputPartitions()</code></pre> <p data-bbox="562 377 763 413">InputPartitions</p> <p data-bbox="562 435 1367 579">Used exclusively when <code>DataSourceV2ScanExec</code> leaf physical operator is requested for the input partitions (and simply delegates to the underlying <code>DataSourceReader</code>) to create the input <code>RDD[InternalRow]</code> (<code>inputRDD</code>)</p>
<code>readSchema</code>	<pre data-bbox="589 653 917 682"><code>StructType readSchema()</code></pre> <p data-bbox="562 743 1298 779">Schema for reading (loading) data from a data source</p> <p data-bbox="562 804 732 840">Used when:</p> <ul data-bbox="589 871 1389 1471" style="list-style-type: none"> <li data-bbox="589 871 1389 1006">• <code>DataSourceV2Relation</code> factory object is requested to create a <code>DataSourceV2Relation</code> (when <code>DataFrameReader</code> is requested to "load" data (as a <code>DataFrame</code>) from a data source with <code>ReadSupport</code>) <li data-bbox="589 1035 1389 1125">• <code>DataSourceV2Strategy</code> execution planning strategy is requested to apply column pruning optimization <li data-bbox="589 1154 1389 1244">• Spark Structured Streaming's <code>MicroBatchExecution</code> stream execution is requested to run a single streaming batch <li data-bbox="589 1273 1389 1385">• Spark Structured Streaming's <code>ContinuousExecution</code> stream execution is requested to run a streaming query in continuous mode <li data-bbox="589 1414 1389 1471">• Spark Structured Streaming's <code>DataStreamReader</code> is requested to "load" data (as a <code>DataFrame</code>)
Note	<p data-bbox="303 1558 1389 1664"><code>DataSourceReader</code> is an <code>Evolving</code> contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.</p> <p data-bbox="303 1693 1129 1729">In other words, using the contract is as "<i>treading on thin ice</i>".</p>

Table 2. DataSourceReaders (Direct Implementations and Extensions Only)

DataSourceReader	Description
ContinuousReader	<code>DataSourceReaders</code> for Continuous Stream Processing in Spark Structured Streaming Consult The Internals of Spark Structured Streaming
MicroBatchReader	<code>DataSourceReaders</code> for Micro-Batch Stream Processing in Spark Structured Streaming Consult The Internals of Spark Structured Streaming
SupportsPushDownFilters	<code>DataSourceReaders</code> that can push down filters to the data source and reduce the size of the data to be read
SupportsPushDownRequiredColumns	<code>DataSourceReaders</code> that can push down required columns to the data source and only read these columns during scan to reduce the size of the data to be read
SupportsReportPartitioning	<code>DataSourceReaders</code> that can report data partitioning and try to avoid shuffle at Spark side
SupportsReportStatistics	<code>DataSourceReaders</code> that can report statistics to Spark
SupportsScanColumnarBatch	<code>DataSourceReaders</code> that can output ColumnarBatch and make the scan faster

SupportsPushDownFilters Contract — Data Source Readers with Filter Pushdown Optimization Support

`SupportsPushDownFilters` is the [extension](#) of the [DataSourceReader contract](#) for [data source readers](#) in [Data Source API V2](#) that support [filter pushdown](#) performance optimization (and hence reduce the size of the data to be read).

Table 1. SupportsPushDownFilters Contract

Method	Description
<code>pushedFilters</code>	<pre>Filter[] pushedFilters()</pre> <p>Data source filters that were pushed down to the data source (in <code>pushFilters</code>)</p> <p>Used exclusively when DataSourceV2Strategy execution planning strategy is executed (on a DataSourceV2Relation logical operator with a <code>SupportsPushDownFilters</code> reader)</p>
<code>pushFilters</code>	<pre>Filter[] pushFilters(Filter[] filters)</pre> <p>Data source filters that need to be evaluated again after scanning (so Spark can plan an extra filter operator)</p> <p>Used exclusively when DataSourceV2Strategy execution planning strategy is executed (on a DataSourceV2Relation logical operator with a <code>SupportsPushDownFilters</code> reader)</p>
Note	<p><code>SupportsPushDownFilters</code> is an Evolving contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.</p> <p>In other words, using the contract is as treading on thin ice.</p>

SupportsPushDownRequiredColumns

SupportsPushDownRequiredColumns is...FIXME

SupportsReportPartitioning

SupportsReportPartitioning is...FIXME

SupportsReportStatistics

SupportsReportStatistics is...FIXME

SupportsScanColumnarBatch

`SupportsScanColumnarBatch` is the [contract](#)...FIXME

```
package org.apache.spark.sql.sources.v2.reader;

public interface SupportsScanColumnarBatch extends DataSourceReader {
    // only required methods that have no implementation
    // the others follow
    List<DataReaderFactory<ColumnarBatch>> createBatchDataReaderFactories();
}
```

Note

`SupportsScanColumnarBatch` is an [Evolving](#) contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.

In other words, using the contract is as treading on thin ice.

Table 1. (Subset of) `SupportsScanColumnarBatch` Contract

Method	Description
<code>createBatchDataReaderFactories</code>	Used when...FIXME

Note

No custom `SupportsScanColumnarBatch` are part of Spark 2.3.

enableBatchRead Method

```
default boolean enableBatchRead()
```

`enableBatchRead` flag is always enabled (i.e. `true`) unless overrode by [custom](#) `SupportsScanColumnarBatches`.

Note

`enableBatchRead` is used when...FIXME

DataSourceWriter Contract

`DataSourceWriter` is the abstraction of data source writers in Data Source API V2 that can [abort](#) or [commit](#) a writing Spark job, [create a DataWriterFactory](#) to be shared among writing Spark tasks and optionally [handle a commit message](#) and [use a CommitCoordinator](#) for writing Spark tasks.

Note

The terms **Spark job** and **Spark task** are really about the low-level Spark jobs and tasks (that you can monitor using web UI for example).

`DataSourceWriter` is used to create a logical [WriteToDataSourceV2](#) and physical [WriteToDataSourceV2Exec](#) operators.

`DataSourceWriter` is created when:

- `DataSourceV2Relation` logical operator is requested to [create one](#)
- `WriteSupport` data source is requested to [create one](#)

Table 1. DataSourceWriter Contract

Method	Description
abort	<pre>void abort(WriterCommitMessage[] messages)</pre> <p>Aborts a writing Spark job Used exclusively when <code>WriteToDataSourceV2Exec</code> physical operator is requested to execute (and an exception was reported)</p>
commit	<pre>void commit(WriterCommitMessage[] messages)</pre> <p>Commits a writing Spark job Used exclusively when <code>WriteToDataSourceV2Exec</code> physical operator is requested to execute (and writing tasks all completed successfully)</p>
	<pre>DataWriterFactory<InternalRow> createWriterFactory()</pre> <p>Creates a DataWriterFactory Used when:</p>

	<ul style="list-style-type: none"> • <code>WriteToDataSourceV2Exec</code> physical operator is requested to <code>execute</code> • Spark Structured Streaming's <code>WriteToContinuousDataSourceExec</code> physical operator is requested to execute • Spark Structured Streaming's <code>MicroBatchWriter</code> is requested to create a <code>DataWriterFactory</code>
<code>onDataWriterCommit</code>	<pre>void onDataWriterCommit(WriterCommitMessage message)</pre> <p>Handles <code>WriterCommitMessage</code> commit message for a single successful writing Spark task</p> <p>Defaults to <i>do nothing</i></p> <p>Used exclusively when <code>WriteToDataSourceV2Exec</code> physical operator is requested to <code>execute</code> (and runs a Spark job with partition writing tasks)</p>
<code>useCommitCoordinator</code>	<pre>boolean useCommitCoordinator()</pre> <p>Controls whether to use a Spark Core <code>OutputCommitCoordinator</code> (<code>true</code>) or not (<code>false</code>) for data writing (to make sure that at most one task for a partition commits)</p> <p>Default: <code>true</code></p> <p>Used exclusively when <code>WriteToDataSourceV2Exec</code> physical operator is requested to <code>execute</code></p>

Table 2. DataSourceWriters (Direct Implementations and Extensions)

DataSourceWriter	Description
<code>MicroBatchWriter</code>	Used in Spark Structured Streaming only for Micro-Batch Stream Processing
<code>StreamWriter</code>	Used in Spark Structured Streaming only (to support epochs)

SessionConfigSupport Contract — Data Sources with Session-SScoped Configuration Options

`SessionConfigSupport` is the contract of [DataSourceV2 data sources](#) in [Data Source API V2](#) that use [custom key prefix for configuration options](#) (i.e. options with `spark.datasource` prefix for the keys in [SQLConf](#)).

With `SessionConfigSupport`, a data source can be configured by additional (session-scoped) configuration options that are specified in [SparkSession](#) that extend user-defined options.

```
String keyPrefix()
```

`keyPrefix` is used exclusively when `DataSourceV2Utils` object is requested to [extract session configuration options](#) (i.e. options with `spark.datasource` prefix for the keys) for [DataSourceV2 data sources](#) with [SessionConfigSupport](#).

`keyPrefix` must not be `null` or an `IllegalArgumentException` is thrown:

```
The data source config key prefix can't be null.
```

InputPartition Contract

`InputPartition` is the abstraction of input partitions in Data Source API V2 that can create an `InputPartitionReader` and optionally specify preferred locations.

`InputPartition` is also a Java `Serializable`.

`InputPartition` is associated with `DataSourceReader` abstraction and its extension `SupportsScanColumnarBatch`.

`InputPartition` is used for the following:

- Creating `DataSourceRDD`, `DataSourceRDDPartition`, `ContinuousDataSourceRDD` and `ContinuousDataSourceRDDPartition`
- Requesting `DataSourceV2ScanExec` physical operator for the `partitions` or `batchPartitions` (of the input RDD)

Note

It appears that all concrete `input partitions` are used in Spark Structured Streaming only.

Table 1. InputPartition Contract

Method	Description
<code>createPartitionReader</code>	<pre>InputPartitionReader<T> createPartitionReader()</pre> <p>Creates an <code>InputPartitionReader</code></p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataSourceRDD</code> is requested to compute a partition • <code>ContinuousQueuedDataReader</code> is created
<code>preferredLocations</code>	<pre>String[] preferredLocations()</pre> <p>Specifies the preferred locations (executor hosts)</p> <p>Default: (empty)</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataSourceRDD</code> is requested for the preferred locations • <code>ContinuousDataSourceRDD</code> is requested for the preferred locations

Table 2. InputPartitions (Direct Implementations and Extensions Only)

InputPartition	Description
ContinuousInputPartition	InputPartitions for Continuous Stream Processing in Spark Structured Streaming Consult The Internals of Spark Structured Streaming
ContinuousMemoryStreamInputPartition	Used in Spark Structured Streaming
KafkaMicroBatchInputPartition	Used in Spark Structured Streaming
MemoryStreamInputPartition	Used in Spark Structured Streaming
RateStreamMicroBatchInputPartition	Used in Spark Structured Streaming
TextSocketContinuousInputPartition	Used in Spark Structured Streaming
Anonymous	Used in Spark Structured Streaming

InputPartitionReader Contract

`InputPartitionReader` is the abstraction of input partition readers in Data Source API V2 that can proceed to the next record and get the current record.

`InputPartitionReader` is also a Java [Closeable](#).

`InputPartitionReader` is associated with two other abstractions: `InputPartition` and `ContinuousInputPartition` that are responsible for creating [concrete InputPartitionReaders](#).

Note	It appears that all concrete input partition readers are used in Spark Structured Streaming only.
------	---

Table 1. InputPartitionReader Contract

Method	Description
<code>get</code>	<p style="margin-top: 0;"><code>T get()</code></p> <p>Gets the current record</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataSourceRDD</code> is requested to compute a partition • <code>DataReaderThread</code> is requested to run (<i>start up</i>)
<code>next</code>	<p style="margin-top: 0;"><code>boolean next() throws IOException</code></p> <p>Proceeds to the next record if available (<code>true</code>)</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataSourceRDD</code> is requested to compute a partition • <code>DataReaderThread</code> is requested to run (<i>start up</i>)

Table 2. InputPartitionReaders (Direct Implementations and Extensions Only)

InputPartitionReader	Description
ContinuousInputPartitionReader	Extension that is used in Spark Structured Streaming for Continuous Stream Processing
KafkaMicroBatchInputPartitionReader	Used in Spark Structured Streaming for Kafka Data Source
Anonymous	Used in Spark Structured Streaming for Memory Data Source
RateStreamMicroBatchInputPartitionReader	Used in Spark Structured Streaming for Rate Data Source
Anonymous	Used in Spark Structured Streaming for Text Socket Data Source

DataWriter

`DataWriter` is...FIXME

DataWriterFactory

`DataWriterFactory` is a [contract](#)...FIXME

```
package org.apache.spark.sql.sources.v2.writer;

public interface DataWriterFactory<T> extends Serializable {
    DataWriter<T> createDataWriter(int partitionId, int attemptNumber);
}
```

Note

`DataWriterFactory` is an [Evolving](#) contract that is evolving towards becoming a stable API, but is not a stable API yet and can change from one feature release to another release.

In other words, using the contract is as treading on thin ice.

Table 1. DataWriterFactory Contract

Method	Description
<code>createDataWriter</code>	<p>Gives the DataWriter for a partition ID and attempt number</p> <p>Used when:</p> <ul style="list-style-type: none"> • InternalRowDataWriterFactory is requested to createDataWriter • DataWritingSparkTask is requested to run and runContinuous

InternalRowDataWriterFactory

InternalRowDataWriterFactory is...FIXME

createDataWriter Method

```
createDataWriter(partitionId: Int, attemptNumber: Int): DataWriter[InternalRow]
```

Note

createDataWriter is part of [DataWriterFactory Contract](#) to...FIXME.

createDataWriter ...FIXME

DataSourceV2StringFormat

DataSourceV2StringFormat is...FIXME

DataSourceRDD — Input RDD Of DataSourceV2ScanExec Physical Operator

`DataSourceRDD` acts as a thin adapter between Spark SQL's [Data Source API V2](#) and Spark Core's `RDD` API.

`DataSourceRDD` is an `RDD` that is [created](#) exclusively when `DataSourceV2ScanExec` physical operator is requested for the [input RDD](#) (when `WholeStageCodegenExec` physical operator is [executed](#)).

`DataSourceRDD` uses `DataSourceRDDPartition` when requested for the [partitions](#) (that is a mere wrapper of the `InputPartitions`).

`DataSourceRDD` takes the following to be created:

- Spark Core's `SparkContext`
- [InputPartitions](#) (`Seq[InputPartition[T]]`)

`DataSourceRDD` as a Spark Core `RDD` overrides the following methods:

- [getPartitions](#)
- [compute](#)
- [getPreferredLocations](#)

Requesting Preferred Locations (For Partition) — `getPreferredLocations` Method

```
getPreferredLocations(split: Partition): Seq[String]
```

Note	<code>getPreferredLocations</code> is part of Spark Core's <code>RDD</code> Contract to...FIXME.
------	--

`getPreferredLocations` simply requests the given `split` `DataSourceRDDPartition` for the `InputPartition` that in turn is requested for the [preferred locations](#).

RDD Partitions — `getPartitions` Method

```
getPartitions: Array[Partition]
```

Note	<code>getPartitions</code> is part of Spark Core's <code>RDD</code> Contract to...FIXME
------	---

`getPartitions` simply creates a [DataSourceRDDPartition](#) for every [InputPartition](#).

Computing Partition (in TaskContext) — `compute` Method

```
compute(split: Partition, context: TaskContext): Iterator[T]
```

Note	<code>compute</code> is part of Spark Core's <code>RDD</code> Contract to compute a partition (in a <code>TaskContext</code>).
------	---

`compute` requests the input [DataSourceRDDPartition](#) (the `split` partition) for the [InputPartition](#) that in turn is requested to [create an InputPartitionReader](#).

`compute` registers a Spark Core `TaskCompletionListener` that requests the [InputPartitionReader](#) to close when a task completes.

`compute` returns a Spark Core `InterruptibleIterator` that requests the [InputPartitionReader](#) to [proceed to the next record](#) (when requested to `hasNext`) and [return the current record](#) (when `next`).

DataSourceRDDPartition

`DataSourceRDDPartition` is a Spark Core Partition of `DataSourceRDD` and Spark Structured Streaming's `ContinuousDataSourceRDD` RDDs.

`DataSourceRDDPartition` is created when:

- `DataSourceRDD` and Spark Structured Streaming's `ContinuousDataSourceRDD` are requested for partitions
- `DataSourceRDD` and Spark Structured Streaming's `ContinuousDataSourceRDD` are requested to compute a partition
- `DataSourceRDD` and Spark Structured Streaming's `ContinuousDataSourceRDD` are requested for preferred locations

`DataSourceRDDPartition` takes the following when created:

- Partition index
- `InputPartition`

DataWritingSparkTask Partition Processing Function

`DataWritingSparkTask` is the **partition processing function** that `writeToDataSourceV2Exec` physical operator uses to [schedule a Spark job](#) when requested to [execute](#).

<p>Note</p> <p>The <code>DataWritingSparkTask</code> partition processing function is executed on executors.</p>
<p>Tip</p> <p>Enable <code>INFO</code> or <code>ERROR</code> logging levels for <code>org.apache.spark.sql.execution.datasources.v2.DataWritingSparkTask</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.datasources.v2.DataWritingSparkTask=</pre> <p>Refer to Logging.</p>

Running Partition Processing Function — `run` Method

```
run(
  writeTask: DataWriterFactory[InternalRow],
  context: TaskContext,
  iter: Iterator[InternalRow],
  useCommitCoordinator: Boolean): WriterCommitMessage
```

`run` requests the given `TaskContext` for the IDs of the stage, the stage attempt, the partition, the task attempt, and how many times the task may have been attempted (default `0`).

`run` also requests the given `TaskContext` for the epoch ID (that is `streaming.sql.batchId` local property) or defaults to `0`.

`run` requests the given `DataWriterFactory` to [create a DataWriter](#) (with the partition, task and epoch IDs).

For every row in the partition (in the given `Iterator[InternalRow]`), `run` requests the `DataWriter` to [write the row](#).

Once all the rows have been written successfully, `run` requests the `DataWriter` to [commit the write task](#) (with or without requesting the `OutputCommitCoordinator` for authorization) that gives the final `WriterCommitMessage`.

In the end, `run` prints out the following INFO message to the logs:

```
Committed partition [partitionId] (task [taskId], attempt [attemptId] stage [stageId].[stageAttempt])
```

In case of any errors, `run` prints out the following ERROR message to the logs:

```
Aborting commit for partition [partitionId] (task [taskId], attempt [attemptId] stage [stageId].[stageAttempt])
```

`run` then requests the `DataWriter` to [abort the write task](#).

In the end, `run` prints out the following ERROR message to the logs:

```
Aborted commit for partition [partitionId] (task [taskId], attempt [attemptId] stage [stageId].[stageAttempt])
```

Note	<code>run</code> is used exclusively when <code>WriteToDataSourceV2Exec</code> physical operator is requested to execute (and schedules a Spark job).
------	--

useCommitCoordinator Flag Enabled

With the given `useCommitCoordinator` flag enabled (the default for most [DataSourceWriters](#)), `run` requests the `sparkEnv` for the `OutputCommitCoordinator` that is then requested whether to commit the write task output or not (`canCommit`).

Tip	Read up on OutputCommitCoordinator in the Mastering Apache Spark .
-----	--

If authorized, `run` prints out the following INFO message to the logs:

```
Commit authorized for partition [partitionId] (task [taskId], attempt [attemptId] stage [stageId].[stageAttempt])
```

In the end, `run` requests the `DataWriter` to [commit the write task](#).

If not authorized, `run` prints out the following INFO message to the logs and throws a `CommitDeniedException`.

```
Commit denied for partition [partitionId] (task [taskId], attempt [attemptId] stage [stageId]
]. [stageAttempt])
```

useCommitCoordinator Flag Disabled

With the given `useCommitCoordinator` flag disabled, `run` prints out the following INFO message to the logs:

```
Writer for partition [partitionId] is committing.
```

In the end, `run` requests the `DataWriter` to [commit the write task](#).

DataSourceV2Utils Helper Object

`DataSourceV2Utils` is a helper object that is used exclusively to [extract session configuration options](#) (i.e. options with `spark.datasource` prefix for the keys in `SQLConf`) for `DataSourceV2` data sources with [SessionConfigSupport](#) in Data Source API V2.

```
extractSessionConfigs(  
    ds: DataSourceV2,  
    conf: SQLConf): Map[String, String]
```

Note	<code>extractSessionConfigs</code> supports data sources with SessionConfigSupport only.
------	--

`extractSessionConfigs` requests the `SessionConfigSupport` data source for the [custom key prefix for configuration options](#) that is used to find all configuration options with the keys in the format of `spark.datasource.[keyPrefix]` in the given `SQLConf`.

`extractSessionConfigs` returns the matching keys with the `spark.datasource.[keyPrefix]` prefix removed (i.e. `spark.datasource.keyPrefix.k1` becomes `k1`).

Note	<p><code>extractSessionConfigs</code> is used when:</p> <ul style="list-style-type: none"> • <code>DataFrameReader</code> is requested to "load" data as a <code>DataFrame</code> (for a <code>DataSourceV2</code> with ReadSupport) • <code>DataFrameWriter</code> is requested to saves a DataFrame to a data source (for a <code>DataSourceV2</code> with WriteSupport) • Spark Structured Streaming's <code>DataStreamReader</code> is requested to load input data stream (for data sources with <code>MicroBatchReadSupport</code> or <code>ContinuousReadSupport</code>) • Spark Structured Streaming's <code>DataStreamWriter</code> is requested to start a streaming query (with data sources with <code>StreamWriteSupport</code>)
------	---

- `DataFrameReader` is requested to "load" data as a `DataFrame` (for a `DataSourceV2` with [ReadSupport](#))
- `DataFrameWriter` is requested to [saves a DataFrame to a data source](#) (for a `DataSourceV2` with [WriteSupport](#))
- Spark Structured Streaming's `DataStreamReader` is requested to load input data stream (for data sources with `MicroBatchReadSupport` or `ContinuousReadSupport`)
- Spark Structured Streaming's `DataStreamWriter` is requested to start a streaming query (with data sources with `StreamWriteSupport`)

DataSource — Pluggable Data Provider Framework

`DataSource` is one of the main parts of **Data Source API** in Spark SQL (together with `DataFrameReader` for loading datasets, `DataFrameWriter` for saving datasets and `StreamSourceProvider` for creating streaming sources).

`DataSource` models a **pluggable data provider framework** with the [extension points](#) for Spark SQL integrators to expand the list of supported external data sources in Spark SQL.

`DataSource` takes a list of [file system paths that hold data](#). The list is empty by default, but can be different per data source:

- The [location URI](#) of a `HiveTableRelation` (when `HiveMetastoreCatalog` is requested to convert a `HiveTableRelation` to a `LogicalRelation`)
- The table name of a `UnresolvedRelation` (when `ResolveSQLOnFile` logical evaluation rule is executed)
- The files in a directory when Spark Structured Streaming's `FileStreamSource` is requested for batches

`DataSource` is [created](#) when:

- `DataStreamWriter` is requested to save to a data source (per [Data Source V1 contract](#))
- `FindDataSourceTable` and `ResolveSQLOnFile` logical evaluation rules are executed
- `CreateDataSourceTableCommand`, `CreateDataSourceTableAsSelectCommand`, `InsertIntoDataSourceDirCommand`, `CreateTempViewUsing` are executed
- `HiveMetastoreCatalog` is requested to `convertToLogicalRelation`
- Spark Structured Streaming's `FileStreamSource`, `DataStreamReader` and `DataStreamWriter`

Table 1. DataSource's Provider (and Format) Contracts

Extension Point	Description
CreatableRelationProvider	Data source that saves the result of a structured query per save mode and returns the schema
FileFormat	Used in: <ul style="list-style-type: none"> • <code>sourceSchema</code> for streamed reading • <code>write</code> for writing a <code>DataFrame</code> to a <code>DataSource</code> (as part of creating a table as select)
RelationProvider	Data source that supports schema inference and can be accessed using SQL's <code>USING</code> clause
SchemaRelationProvider	Data source that requires a user-defined schema
StreamSourceProvider	Used in: <ul style="list-style-type: none"> • <code>sourceSchema</code> and <code>createSource</code> for streamed reading • <code>createSink</code> for streamed writing • <code>resolveRelation</code> for resolved <code>BaseRelation</code>.

As a user, you interact with `DataSource` by `DataFrameReader` (when you execute `spark.read` or `spark.readStream`) or SQL's `CREATE TABLE USING`.

```
// Batch reading
val people: DataFrame = spark.read
  .format("csv")
  .load("people.csv")

// Streamed reading
val messages: DataFrame = spark.readStream
  .format("kafka")
  .option("subscribe", "topic")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .load
```

`DataSource` uses a `SparkSession`, a class name, a collection of `paths`, optional user-specified `schema`, a collection of partition columns, a bucket specification, and configuration options.

Note	Data source is also called a table provider .
------	--

When requested to [resolve a batch \(non-streaming\) FileFormat](#), `DataSource` creates a [HadoopFsRelation](#) with the optional [bucketing specification](#).

Table 2. DataSource's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>providingClass</code>	The Java class (<code>java.lang.Class</code>) that...FIXME Used when...FIXME
<code>sourceInfo</code>	<code>SourceInfo</code> Used when...FIXME
<code>caseInsensitiveOptions</code>	FIXME Used when...FIXME
<code>equality</code>	FIXME Used when...FIXME
<code>backwardCompatibilityMap</code>	FIXME Used when...FIXME

Writing Data to Data Source per Save Mode Followed by Reading Rows Back (as BaseRelation) — `writeAndRead` Method

```
writeAndRead(  
    mode: SaveMode,  
    data: DataFrame): BaseRelation
```

`writeAndRead` ...FIXME

Note	<code>writeAndRead</code> is also known as Create Table As Select (CTAS) query.
------	--

Note	<code>writeAndRead</code> is used exclusively when CreateDataSourceTableAsSelectCommand logical command is executed.
------	--

Writing DataFrame to Data Source Per Save Mode — `write` Method

```
write(mode: SaveMode, data: DataFrame): BaseRelation
```

`write` writes the result of executing a structured query (as `DataFrame`) to a data source per `save` mode .

Internally, `write` looks up the data source and branches off per `providingClass`.

Table 3. write's Branches per Supported providingClass (in execution order)

providingClass	Description
<code>CreatableRelationProvider</code>	Executes <code>CreatableRelationProvider.createRelation</code>
<code>FileFormat</code>	<code>writelnFileFormat</code>
<i>others</i>	Reports a <code>RuntimeException</code>

Note	<code>write</code> does not support the internal <code>CalendarIntervalType</code> in the schema of <code>data</code> <code>DataFrame</code> and throws a <code>AnalysisException</code> when there is one.
Note	<code>write</code> is used exclusively when <code>SaveIntoDataSourceCommand</code> is executed.

writelnFileFormat Internal Method

Caution	FIXME
---------	-------

For `FileFormat` data sources, `write` takes all `paths` and `path` option and makes sure that there is only one.

Note	<code>write</code> uses Hadoop's <code>Path</code> to access the <code>FileSystem</code> and calculate the qualified output path.
------	---

`write` requests `PartitioningUtils` to `validatePartitionColumn`.

When appending to a table, ...FIXME

In the end, `write` (for a `FileFormat` data source) prepares a `InsertIntoHadoopFsRelationCommand` logical plan with executes it.

Caution	FIXME Is <code>toRdd</code> a job execution?
---------	--

createSource Method

<code>createSource(metadataPath: String): Source</code>

Caution	FIXME
---------	-------

createSink Method

Caution

FIXME

sourceSchema Internal Method

```
sourceSchema(): SourceInfo
```

`sourceSchema` returns the name and [schema](#) of the data source for streamed reading.

Caution

FIXME Why is the method called? Why does this bother with streamed reading and data sources?!

It supports two class hierarchies, i.e. [FileFormat](#) and Structured Streaming's [StreamSourceProvider](#) data sources.

Internally, `sourceSchema` first creates an instance of the data source and...

Caution

FIXME Finish...

For Structured Streaming's [StreamSourceProvider](#) data sources, `sourceSchema` relays calls to `StreamSourceProvider.sourceSchema`.

For [FileFormat](#) data sources, `sourceSchema` makes sure that `path` option was specified.

Tip

`path` is looked up in a case-insensitive way so `paTh` and `PATH` and `pATH` are all acceptable. Use the lower-case version of `path`, though.

Note

`path` can use [glob pattern](#) (not regex syntax), i.e. contain any of `{ } [] * ? \` characters.

It checks whether the path exists if a glob pattern is not used. In case it did not exist you will see the following [AnalysisException](#) exception in the logs:

```

scala> spark.read.load("the.file.does.not.exist.parquet")
org.apache.spark.sql.AnalysisException: Path does not exist: file:/Users/jacek/dev/oss
/spark/the.file.does.not.exist.parquet;
   at org.apache.spark.sql.execution.datasources.DataSource$$anonfun$12.apply(DataSourc
e.scala:375)
   at org.apache.spark.sql.execution.datasources.DataSource$$anonfun$12.apply(DataSourc
e.scala:364)
   at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:2
41)
   at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:2
41)
   at scala.collection.immutable.List.foreach(List.scala:381)
   at scala.collection.TraversableLike$class.flatMap(TraversableLike.scala:241)
   at scala.collection.immutable.List.flatMap(List.scala:344)
   at org.apache.spark.sql.execution.datasources.DataSource.resolveRelation(DataSource.
scala:364)
   at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:149)
   at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:132)
... 48 elided

```

If `spark.sql.streaming.schemaInference` is disabled and the data source is different than `TextFileFormat`, and the input `userSpecifiedSchema` is not specified, the following `IllegalArgumentException` exception is thrown:

Schema must be specified when creating a streaming source DataFrame. If some files already exist in the directory, then depending on the file format you may be able to create a static DataFrame on that directory with '`spark.read.load(directory)`' and infer schema from it.

Caution	FIXME I don't think the exception will ever happen for non-streaming sources since the schema is going to be defined earlier. When?
---------	---

Eventually, it returns a `SourceInfo` with `FileSource[path]` and the schema (as calculated using the `inferFileFormatSchema` internal method).

For any other data source, it throws `UnsupportedOperationException` exception:

```
Data source [className] does not support streamed reading
```

Note	<code>sourceSchema</code> is used exclusively when <code>DataSource</code> is requested for the <code>sourceInfo</code> .
------	---

inferFileFormatSchema Internal Method

```
inferFileFormatSchema(format: FileFormat): StructType
```

`inferFileFormatSchema` private method computes (aka *infers*) schema (as `StructType`). It returns `userSpecifiedSchema` if specified or uses `FileFormat.inferSchema`. It throws a `AnalysisException` when is unable to infer schema.

It uses `path` option for the list of directory paths.

Note

It is used by `DataSource.sourceSchema` and `DataSource.createSource` when `FileFormat` is processed.

Resolving Relation (Creating BaseRelation) — `resolveRelation` Method

```
resolveRelation(checkFilesExist: Boolean = true): BaseRelation
```

`resolveRelation` resolves (i.e. creates) a `BaseRelation`.

Internally, `resolveRelation` tries to create an instance of the `providingClass` and branches off per its type and whether the optional `user-specified schema` was specified or not.

Table 4. Resolving BaseRelation per Provider and User-Specified Schema

Provider	Behaviour
<code>SchemaRelationProvider</code>	Executes <code>SchemaRelationProvider.createRelation</code> with the provided schema
<code>RelationProvider</code>	Executes <code>RelationProvider.createRelation</code>
<code>FileFormat</code>	Creates a <code>HadoopFsRelation</code>

	<p><code>resolveRelation</code> is used when:</p> <ul style="list-style-type: none"> • <code>DataSource</code> is requested to write and read the result of a structured query (only when <code>providingClass</code> is a FileFormat) • <code>DataFrameReader</code> is requested to load data from a data source that supports multiple paths • <code>TextInputCSVDataSource</code> and <code>TextInputJsonDataSource</code> are requested to infer schema
Note	<ul style="list-style-type: none"> • <code>CreateDataSourceTableCommand</code> runnable command is executed • <code>CreateTempViewUsing</code> logical command is requested to run • <code>FindDataSourceTable</code> is requested to readDataSourceTable • <code>ResolvesQLOnFile</code> is requested to convert a logical plan (when <code>providingClass</code> is a FileFormat) • <code>HiveMetastoreCatalog</code> is requested for convertToLogicalRelation • Structured Streaming's <code>FileStreamSource</code> creates batches of records

buildStorageFormatFromOptions Method

```
buildStorageFormatFromOptions(options: Map[String, String]): CatalogStorageFormat
```

`buildStorageFormatFromOptions` ...FIXME

Note	<code>buildStorageFormatFromOptions</code> is used when...FIXME
------	---

Creating DataSource Instance

`DataSource` takes the following when created:

- [SparkSession](#)
- Name of the provider class (aka *input data source format*)
- A list of file system paths that hold data (default: empty)
- (optional) User-specified `schema` (default: `None`, i.e. undefined)
- (optional) Names of the partition columns (default: empty)
- Optional [bucketing specification](#) (default: `None`)
- Options (default: empty)

- (optional) `CatalogTable` (default: `None`)

`DataSource` initializes the [internal registries and counters](#).

Looking Up Class By Name Of Data Source Provider

— `lookupDataSource` Method

```
lookupDataSource(  
    provider: String,  
    conf: SQLConf): Class[_]
```

`lookupDataSource` looks up the class name in the [backwardCompatibilityMap](#) and then replaces the class name exclusively for the `orc` provider per [spark.sql.orc.impl](#) internal configuration property:

- For `hive` (default), `lookupDataSource` uses `org.apache.spark.sql.hive.orc.OrcFileFormat`
- For `native`, `lookupDataSource` uses the canonical class name of [OrcFileFormat](#), i.e. `org.apache.spark.sql.execution.datasources.orc.OrcFileFormat`

With the provider's class name (aka `provider1` internally) `lookupDataSource` assumes another name variant of format `[provider1].DefaultSource` (aka `provider2` internally).

`lookupDataSource` then uses Java's [ServiceLoader](#) to find all [DataSourceRegister](#) provider classes on the CLASSPATH.

`lookupDataSource` filters out the `DataSourceRegister` provider classes (by their [alias](#)) that match the `provider1` (case-insensitive), e.g. `parquet` or `kafka`.

If a single provider class was found for the alias, `lookupDataSource` simply returns the provider class.

If no `DataSourceRegister` could be found by the short name (alias), `lookupDataSource` considers the names of the format provider as the fully-qualified class names and tries to load them instead (using Java's [ClassLoader.loadClass](#)).

Note	You can reference your own custom <code>DataSource</code> in your code by DataFrameWriter.format method which is the alias or a fully-qualified class name.
------	---

Caution	FIXME Describe the other cases (orc and avro)
---------	---

If no provider class could be found, `lookupDataSource` throws a `RuntimeException`:

```
java.lang.ClassNotFoundException: Failed to find data source:  
[provider1]. Please find packages at  
http://spark.apache.org/third-party-projects.html
```

If however, `lookupDataSource` found multiple registered aliases for the provider name...

FIXME

Creating Logical Command for Writing (for CreatableRelationProvider and FileFormat Data Sources) — `planForWriting` Method

```
planForWriting(mode: SaveMode, data: LogicalPlan): LogicalPlan
```

`planForWriting` creates an instance of the `providingClass` and branches off per its type as follows:

- For a `CreatableRelationProvider`, `planForWriting` creates a `SaveIntoDataSourceCommand` (with the input `data` and `mode`, the `CreatableRelationProvider` data source and the `caseInsensitiveOptions`)
- For a `FileFormat`, `planForWriting` `planForWritingFileFormat` (with the `FileFormat` format and the input `mode` and `data`)
- For other types, `planForWriting` simply throws a `RuntimeException`:

[`providingClass`] does not allow create table as select.

Note

`planForWriting` is used when:

- `DataFrameWriter` is requested to `saveToV1Source` (when `DataFrameWriter` is requested to `save the result of a structured query (a DataFrame) to a data source` for `DataSourceV2` with no `writeSupport` and non-`DataSourceV2` writers)
- `InsertIntoDataSourceDirCommand` logical command is executed

Planning for Writing (Using FileFormat) — `planForWritingFileFormat` Internal Method

```
planForWritingFileFormat(
    format: FileFormat,
    mode: SaveMode,
    data: LogicalPlan): InsertIntoHadoopFsRelationCommand
```

`planForWritingFileFormat` takes the `paths` and the `path` option (from the `caseInsensitiveOptions`) together and (assuming that there is only one path available among the paths combined) creates a fully-qualified HDFS-compatible output path for writing.

Note

`planForWritingFileFormat` uses Hadoop HDFS's `Path` to requests for the `FileSystem` that owns it (using [Hadoop Configuration](#)).

`planForWritingFileFormat` uses the [PartitioningUtils](#) helper object to validate partition `columns` in the `partitionColumns`.

In the end, `planForWritingFileFormat` returns a new [InsertIntoHadoopFsRelationCommand](#).

When the number of the `paths` is different than `1`, `planForWritingFileFormat` throws an `IllegalArgumentException`:

```
Expected exactly one path to be specified, but got: [allPaths]
```

Note

`planForWritingFileFormat` is used when `DataSource` is requested to [write data to a data source per save mode followed by reading rows back](#) (when `CreateDataSourceTableAsSelectCommand` logical command is executed) and for the [logical command for writing](#).

getOrInferFileFormatSchema Internal Method

```
getOrInferFileFormatSchema(
    format: FileFormat,
    fileStatusCache: FileStatusCache = NoopCache): (StructType, StructType)
```

`getOrInferFileFormatSchema` ...FIXME

Note

`getOrInferFileFormatSchema` is used when `DataSource` is requested for the `sourceSchema` and to `resolveRelation`.

Custom Data Source Formats

Caution	FIXME
---------	-------

See [spark-mf-format](#) project at GitHub for a complete solution.

CreatableRelationProvider Contract — Data Sources That Write Rows Differently Per Save Mode

`CreatableRelationProvider` is the abstraction of data source providers that can write the rows of a structured query (a `DataFrame`) differently per save mode.

Table 1. `CreatableRelationProvider` Contract

Method	Description
<code>createRelation</code>	<pre>createRelation(sqlContext: SQLContext, mode: SaveMode, parameters: Map[String, String], data: DataFrame): BaseRelation</pre> <p>Creates a <code>BaseRelation</code> that represents the rows of a structured query (a <code>DataFrame</code>) saved to an external data source differently per <code>SaveMode</code></p> <p>The save mode specifies what should happen when the target relation (destination) already exists.</p> <p>Used when:</p> <ul style="list-style-type: none"> <code>DataSource</code> is requested to write data to a data source per save mode followed by reading the rows back (when <code>CreateDataSourceTableAsSelectCommand</code> logical command is executed) <code>SaveIntoDataSourceCommand</code> logical command is executed

Table 2. `CreatableRelationProviders`

<code>CreatableRelationProvider</code>	Description
<code>ConsoleSinkProvider</code>	Data source provider for <code>Console</code> data source
<code>JdbcRelationProvider</code>	Data source provider for <code>JDBC</code> data source
<code>KafkaSourceProvider</code>	Data source provider for <code>Kafka</code> data source

DataSourceRegister Contract — Registering Data Source Format

`DataSourceRegister` is a [contract](#) to register a `DataSource` provider under `shortName` alias (so it can be [looked up](#) by the alias not its fully-qualified class name).

```
package org.apache.spark.sql.sources

trait DataSourceRegister {
  def shortName(): String
}
```

Data Source Format Discovery — Registering Data Source By Short Name (Alias)

Caution

FIXME Describe how Java's [ServiceLoader](#) works to find all `DataSourceRegister` provider classes on the CLASSPATH.

Any `DataSourceRegister` has to register itself in `META-INF/services/org.apache.spark.sql.sources.DataSourceRegister` file to...FIXME

RelationProvider Contract — Relation Providers With Schema Inference

`RelationProvider` is the [contract](#) of [BaseRelation](#) providers that [create a relation with schema inference](#).

Note

Schema inference is also called **schema discovery**.

The requirement of not specifying a user-defined schema or having one that does not match the relation is enforced when `DataSource` is requested for a [BaseRelation](#) for a given data source format. If specified and does not match, `DataSource` throws a `AnalysisException`:

```
[className] does not allow user-specified schemas.
```

```
package org.apache.spark.sql.sources

trait RelationProvider {
  def createRelation(
    sqlContext: SQLContext,
    parameters: Map[String, String]): BaseRelation
}
```

Table 1. RelationProvider Contract

Method	Description
<code>createRelation</code>	Creates a BaseRelation for loading data from an external data source Used exclusively when <code>DataSource</code> is requested for a BaseRelation for a given data source format (and no user-defined schema or the user-defined schema matches schema of the <code>BaseRelation</code>)

Table 2. RelationProviders

RelationProvider	Description
JdbcRelationProvider	
KafkaSourceProvider	

Tip

Use [SchemaRelationProvider](#) for relation providers that require a user-defined schema.

SchemaRelationProvider Contract — Relation Providers With Mandatory User-Defined Schema

`SchemaRelationProvider` is the [contract](#) of `BaseRelation` providers that require a user-defined schema while creating a relation.

The requirement of specifying a user-defined schema is enforced when `DataSource` is requested for a `BaseRelation` for a given data source format. If not specified, `DataSource` throws a `AnalysisException`:

```
A schema needs to be specified when using [className].
```

```
package org.apache.spark.sql.sources

trait SchemaRelationProvider {
  def createRelation(
    sqlContext: SQLContext,
    parameters: Map[String, String],
    schema: StructType): BaseRelation
}
```

Table 1. SchemaRelationProvider Contract

Method	Description
<code>createRelation</code>	Creates a <code>BaseRelation</code> for the user-defined schema Used exclusively when <code>DataSource</code> is requested for a <code>BaseRelation</code> for a given data source format
Note	There are no known direct implementation of PrunedFilteredScan Contract in Spark SQL.
Tip	Use RelationProvider for data source providers with schema inference.
Tip	Use both <code>SchemaRelationProvider</code> and RelationProvider if a data source should support both schema inference and user-defined schemas.

BaseRelation — Collection of Tuples with Schema

`BaseRelation` is the contract of [relations](#) (aka *collections of tuples*) with a known [schema](#).

Note	"Data source", "relation" and "table" are often used as synonyms.
------	---

```
package org.apache.spark.sql.sources

abstract class BaseRelation {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def schema: StructType
    def sqlContext: SQLContext
}
```

Table 1. (Subset of) BaseRelation Contract

Method	Description
<code>schema</code>	StructType that describes the schema of tuples
<code>sqlContext</code>	SQLContext

`BaseRelation` is "created" when `DataSource` is requested to [resolve a relation](#).

`BaseRelation` is transformed into a `DataFrame` when `SparkSession` is requested to [create a DataFrame](#).

`BaseRelation` uses [needConversion](#) flag to control type conversion of objects inside `Rows` to Catalyst types, e.g. `java.lang.String` to `UTF8String`.

Note	It is recommended that custom data sources (outside Spark SQL) should leave needConversion flag enabled, i.e. <code>true</code> .
------	---

`BaseRelation` can optionally give an [estimated size](#) (in bytes).

Table 2. BaseRelations

BaseRelation	Description
ConsoleRelation	Used in Spark Structured Streaming
HadoopFsRelation	
JDBCRelation	
KafkaRelation	Datasets with records from Apache Kafka

Should JVM Objects Inside Rows Be Converted to Catalyst Types? — `needConversion` Method

```
needConversion: Boolean
```

`needConversion` flag is enabled (`true`) by default.

Note	It is recommended to leave <code>needConversion</code> enabled for data sources outside Spark SQL.
------	--

Note	<code>needConversion</code> is used exclusively when <code>DataSourceStrategy</code> execution planning strategy is <code>executed</code> (and does the <code>RDD</code> conversion from <code>RDD[Row]</code> to <code>RDD[InternalRow]</code>).
------	--

Finding Unhandled Filter Predicates — `unhandledFilters` Method

```
unhandledFilters(filters: Array[Filter]): Array[Filter]
```

`unhandledFilters` returns [Filter predicates](#) that the data source does not support (handle) natively.

Note	<code>unhandledFilters</code> returns the input <code>filters</code> by default as it is considered safe to double evaluate filters regardless whether they could be supported or not.
------	--

Note	<code>unhandledFilters</code> is used exclusively when <code>DataSourceStrategy</code> execution planning strategy is requested to <code>selectFilters</code> .
------	---

Requesting Estimated Size — `sizeInBytes` Method

```
sizeInBytes: Long
```

`sizeInBytes` is the estimated size of a relation (used in query planning).

Note

`sizeInBytes` defaults to `spark.sql.defaultSizeInBytes` internal property (i.e. infinite).

Note

`sizeInBytes` is used exclusively when `LogicalRelation` is requested to `computeStats` (and they are not available in `CatalogTable`).

HadoopFsRelation — Relation for File-Based Data Source

`HadoopFsRelation` is a [BaseRelation](#) and [FileRelation](#).

`HadoopFsRelation` is created when:

- `HiveMetastoreCatalog` is requested to [convertToLogicalRelation](#) (when `RelationConversions` logical evaluation rule is requested to [convert a HiveTableRelation to a LogicalRelation](#) for `parquet` or `native` and `hive` ORC storage formats)
- `DataSource` is requested to [create a BaseRelation](#) (for a non-streaming file-based data source)

The optional [BucketSpec](#) is defined exclusively for a non-streaming file-based data source and used for the following:

- [Output partitioning scheme](#) and [output data ordering](#) of the corresponding `FileSourceScanExec` physical operator
- [DataSourceAnalysis](#) post-hoc logical resolution rule (when executed on a `InsertIntoTable` logical operator over a [LogicalRelation](#) with `HadoopFsRelation` relation)

CAUTION: Demo the different cases when `HadoopFsRelation` is created

```
import org.apache.spark.sql.execution.datasources.{HadoopFsRelation, LogicalRelation}

// Example 1: spark.table for DataSource tables (provider != hive)
import org.apache.spark.sql.catalyst.TableIdentifier
val t1ID = TableIdentifier(tableName = "t1")
spark.sessionState.catalog.dropTable(name = t1ID, ignoreIfNotExists = true, purge = true)
spark.range(5).write.saveAsTable("t1")

val metadata = spark.sessionState.catalog.getTableMetadata(t1ID)
scala> println(metadata.provider.get)
parquet

assert(metadata.provider.get != "hive")

val q = spark.table("t1")
// Avoid dealing with UnresolvedRelations and SubqueryAliases
// Hence going straight for optimizedPlan
val plan1 = q.queryExecution.optimizedPlan

scala> println(plan1.numberedTreeString)
00 Relation[id#7L] parquet
```

```

val LogicalRelation(rel1, _, _, _) = plan1.asInstanceOf[LogicalRelation]
val hadoopFsRel = rel1.asInstanceOf[HadoopFsRelation]

// Example 2: spark.read with format as a `FileFormat`
val q = spark.read.text("README.md")
val plan2 = q.queryExecution.logical

scala> println(plan2.numberedTreeString)
00 Relation[value#2] text

val LogicalRelation(relation, _, _, _) = plan2.asInstanceOf[LogicalRelation]
val hadoopFsRel = relation.asInstanceOf[HadoopFsRelation]

// Example 3: Bucketing specified
val tableName = "bucketed_4_id"
spark
  .range(100000000)
  .write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode("overwrite")
  .saveAsTable(tableName)

val q = spark.table(tableName)
// Avoid dealing with UnresolvedRelations and SubqueryAliases
// Hence going straight for optimizedPlan
val plan3 = q.queryExecution.optimizedPlan

scala> println(plan3.numberedTreeString)
00 Relation[id#52L] parquet

val LogicalRelation(rel3, _, _, _) = plan3.asInstanceOf[LogicalRelation]
val hadoopFsRel = rel3.asInstanceOf[HadoopFsRelation]
val bucketSpec = hadoopFsRel.bucketSpec.get

// Exercise 3: spark.table for Hive tables (provider == hive)

```

Creating HadoopFsRelation Instance

`HadoopFsRelation` takes the following when created:

- Location (as `FileIndex`)
- Partition `schema`
- Data `schema`
- Optional `bucketing specification`
- `FileFormat`

- Options
- [SparkSession](#)

`HadoopFsRelation` initializes the [internal registries and counters](#).

CatalystScan Contract

CatalystScan is...FIXME

InsertableRelation Contract — Non-File-Based Relations with Inserting or Overwriting Data Support

`InsertableRelation` is the [contract](#) of non-file-based [BaseRelations](#) that support [inserting](#) or [overwriting data](#).

```
package org.apache.spark.sql.sources

trait InsertableRelation {
  def insert(data: DataFrame, overwrite: Boolean): Unit
}
```

Table 1. InsertableRelation Contract

Property	Description
<code>insert</code>	Inserts or overwrites data (as DataFrame) in a relation Used exclusively when InsertIntoDataSourceCommand logical command is executed
Note	JDBCRelation is the one and only known direct implementation of InsertableRelation Contract in Spark SQL.

PrunedFilteredScan Contract — Relations with Column Pruning and Filter Pushdown

`PrunedFilteredScan` is the [contract](#) of `BaseRelations` with support for [column pruning](#) (i.e. eliminating unneeded columns) and [filter pushdown](#) (i.e. filtering using selected predicates only).

```
package org.apache.spark.sql.sources

trait PrunedFilteredScan {
  def buildScan(requiredColumns: Array[String], filters: Array[Filter]): RDD[Row]
}
```

Table 1. PrunedFilteredScan Contract

Property	Description
<code>buildScan</code>	<p>Building distributed data scan with column pruning and filter pushdown</p> <p>In other words, <code>buildScan</code> creates a <code>RDD[Row]</code> to represent a distributed data scan (i.e. scanning over data in a relation)</p> <p>Used exclusively when <code>DataSourceStrategy</code> execution planning strategy is requested to plan a LogicalRelation with a PrunedFilteredScan.</p>
Note	<code>PrunedFilteredScan</code> is a "lighter" and stable version of the CatalystScan Contract .
Note	<code>JDBCRelation</code> is the one and only known implementation of the PrunedFilteredScan Contract in Spark SQL.

```

// Use :paste to define MyBaseRelation case class
// BEGIN
import org.apache.spark.sql.sources.PrunedFilteredScan
import org.apache.spark.sql.sources.BaseRelation
import org.apache.spark.sql.types.{StructField, StructType, StringType}
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.sources.Filter
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.Row
case class MyBaseRelation(sqlContext: SQLContext) extends BaseRelation with PrunedFilteredScan {
  override def schema: StructType = StructType(StructField("a", StringType) :: Nil)
  def buildScan(requiredColumns: Array[String], filters: Array[Filter]): RDD[Row] = {
    println(s">>> [buildScan] requiredColumns = ${requiredColumns.mkString(",")}")
    println(s">>> [buildScan] filters = ${filters.mkString(",")}")
    import sqlContext.implicits._
    (0 to 4).toDF.rdd
  }
}
// END
val scan = MyBaseRelation(spark.sqlContext)

import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.execution.datasources.LogicalRelation
val plan: LogicalPlan = LogicalRelation(scan)

scala> println(plan.numberedTreeString)
00 Relation[a#1] MyBaseRelation(org.apache.spark.sql.SQLContext@4a57ad67)

import org.apache.spark.sql.execution.datasources.DataSourceStrategy
val strategy = DataSourceStrategy(spark.sessionState.conf)

val sparkPlan = strategy(plan).head
// >>> [buildScan] requiredColumns = a
// >>> [buildScan] filters =
scala> println(sparkPlan.numberedTreeString)
00 Scan MyBaseRelation(org.apache.spark.sql.SQLContext@4a57ad67) [a#8] PushedFilters: [], ReadSchema: struct<a:string>

```

PrunedScan Contract

PrunedScan is...FIXME

TableScan Contract — Relations with Column Pruning

`TableScan` is the [contract](#) of `BaseRelations` with support for [column pruning](#), i.e. can eliminate unneeded columns before producing an RDD containing all of its tuples as `Row` objects.

```
package org.apache.spark.sql.sources

trait PrunedScan {
  def buildScan(): RDD[Row]
}
```

Table 1. TableScan Contract

Property	Description
<code>buildScan</code>	<p>Building distributed data scan with column pruning</p> <p>In other words, <code>buildScan</code> creates a <code>RDD[Row]</code> to represent a distributed data scan (i.e. scanning over data in a relation).</p> <p>Used exclusively when <code>DataSourceStrategy</code> execution planning strategy is requested to plan a LogicalRelation with a TableScan.</p>
Note	KafkaRelation is the one and only known implementation of the TableScan Contract in Spark SQL.

FileFormatWriter Helper Object

`FileFormatWriter` is a Scala object that allows for [writing the result of a structured query](#).

Tip

Enable `ERROR`, `INFO`, `DEBUG` logging level for `org.apache.spark.sql.execution.datasources.FileFormatWriter` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.datasources.FileFormatWriter=INFO
```

Refer to [Logging](#).

Writing Result of Structured Query (Query Result)

— `write` Method

```
write(  
  sparkSession: SparkSession,  
  plan: SparkPlan,  
  fileFormat: FileFormat,  
  committer: FileCommitProtocol,  
  outputSpec: OutputSpec,  
  hadoopConf: Configuration,  
  partitionColumns: Seq[Attribute],  
  bucketSpec: Option[BucketSpec],  
  statsTrackers: Seq[WriteJobStatsTracker],  
  options: Map[String, String]): Set[String]
```

`write` ...FIXME

Note

`write` is used when:

- `SaveAsHiveFile` is requested to `saveAsHiveFile` (when `InsertIntoHiveDirCommand` and `InsertIntoHiveTable` logical commands are executed)
- `InsertIntoHadoopFsRelationCommand` logical command is executed
- Structured Streaming's `FileStreamSink` is requested to add a streaming batch (`addBatch`)

executeTask Internal Method

```
executeTask(  
    description: WriteJobDescription,  
    sparkStageId: Int,  
    sparkPartitionId: Int,  
    sparkAttemptNumber: Int,  
    committer: FileCommitProtocol,  
    iterator: Iterator[InternalRow]): WriteTaskResult
```

```
executeTask ...FIXME
```

Note	<code>executeTask</code> is used exclusively when <code>FileFormatWriter</code> is requested to write the result of a structured query.
------	---

processStats Internal Method

```
processStats(  
    statsTrackers: Seq[WriteJobStatsTracker],  
    statsPerTask: Seq[Seq[WriteTaskStats]]): Unit
```

```
processStats ...FIXME
```

Note	<code>processStats</code> is used exclusively when <code>FileFormatWriter</code> is requested to write the result of a structured query.
------	--

Data Source Filter Predicate (For Filter Pushdown)

`Filter` is the [contract](#) for [filter predicates](#) that can be pushed down to a relation (aka *data source*).

`Filter` is used when:

- (Data Source API V1) `BaseRelation` is requested for [unhandled filter predicates](#) (and hence `BaseRelation` implementations, i.e. [JDBCRelation](#))
- (Data Source API V1) `PrunedFilteredScan` is requested for [build a scan](#) (and hence `PrunedFilteredScan` implementations, i.e. [JDBCRelation](#))
- `FileFormat` is requested to [buildReader](#) (and hence `FileFormat` implementations, i.e. [OrcFileFormat](#), [CSVFileFormat](#), [JsonFileFormat](#), [TextFileFormat](#) and Spark MLlib's [LibSVMFileFormat](#))
- `FileFormat` is requested to [build a Data Reader](#) with partition column values appended (and hence `FileFormat` implementations, i.e. [OrcFileFormat](#), [ParquetFileFormat](#))
- `RowDataSourceScanExec` is [created](#) (for a [simple text representation](#) (in a query plan tree))
- `DataSourceStrategy` execution planning strategy is requested to [pruneFilterProject](#) (when [executed](#) for [LogicalRelation](#) logical operators with a [PrunedFilteredScan](#) or a [PrunedScan](#))
- `DataSourceStrategy` execution planning strategy is requested to [selectFilters](#)
- `JDBCRDD` is [created](#) and requested to [scanTable](#)
- (Data Source API V2) `supportsPushDownFilters` is requested to [pushFilters](#) and for [pushedFilters](#)

```
package org.apache.spark.sql.sources

abstract class Filter {
    // only required methods that have no implementation
    // the others follow
    def references: Array[String]
}
```

Table 1. Filter Contract

Method	Description
references	<p>Column references, i.e. list of column names that are referenced by a filter</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>Filter</code> is requested to find the column references in a value • And, Or and Not filters are requested for the column references

Table 2. Filters

Filter	Description
And	
EqualNullSafe	
EqualTo	
GreaterThan	
GreaterThanOrEqual	
In	
IsNotNull	
IsNull	
LessThan	
LessThanOrEqual	
Not	
Or	
StringContains	
StringEndsWith	
StringStartsWith	

Finding Column References in Any Value

— `findReferences` Method

```
findReferences(value: Any): Array[String]
```

`findReferences` takes the [references](#) from the `value` filter if it is one or returns an empty array.

Note

`findReferences` is used when [EqualTo](#), [EqualNullSafe](#), [GreaterThan](#), [GreaterThanOrEqual](#), [LessThan](#), [LessThanOrEqual](#) and [In](#) filters are requested for their [column references](#).

FileRelation

`FileRelation` is the [contract](#) of relations that are backed by files.

```
package org.apache.spark.sql.execution

trait FileRelation {
  def inputFiles: Array[String]
}
```

Table 1. FileRelation Contract

Method	Description
<code>inputFiles</code>	The list of files that will be read when scanning the relation. Used exclusively when <code>Dataset</code> is requested to inputFiles

Table 2. FileRelations

FileRelation	Description
HadoopFsRelation	

QueryExecution — Structured Query Execution Pipeline

`QueryExecution` represents the [execution pipeline](#) of a structured query (as a `Dataset`) with execution stages (phases).

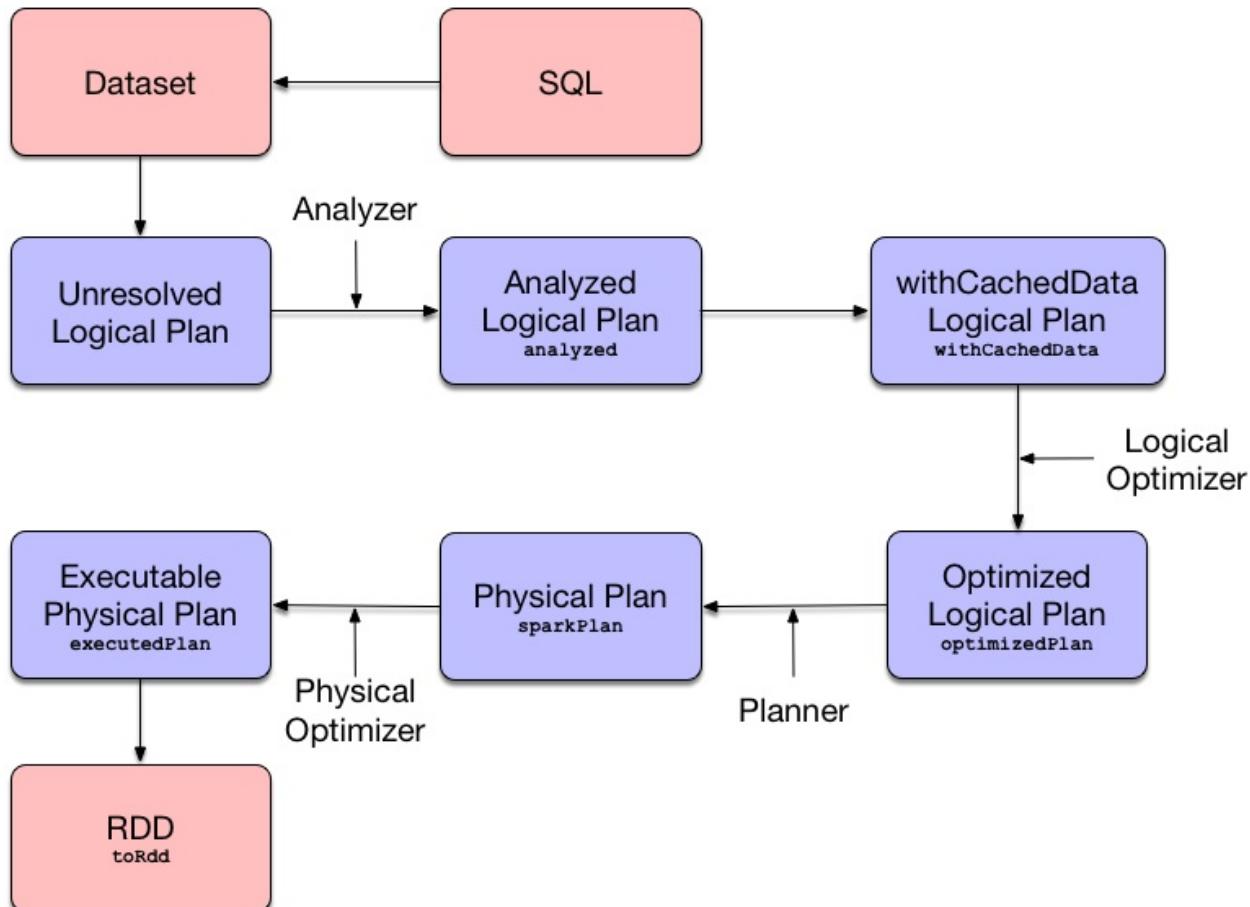


Figure 1. Query Execution — From SQL through Dataset to RDD

Note

When you execute an operator on a `Dataset` it triggers [query execution](#) that gives the good ol' `RDD` of [internal binary rows](#), i.e. `RDD[InternalRow]`, that is Spark's execution plan followed by executing an `RDD` action and so the result of the structured query.

You can access the `QueryExecution` of a `Dataset` using `queryExecution` attribute.

```

val ds: Dataset[Long] = ...
val queryExec = ds.queryExecution
  
```

`QueryExecution` is the result of [executing a LogicalPlan in a SparkSession](#) (and so you could create a `Dataset` from a [logical operator](#) or use the `QueryExecution` after executing a logical operator).

```
val plan: LogicalPlan = ...
val qe = new QueryExecution(sparkSession, plan)
```

Table 1. QueryExecution's Properties (aka QueryExecution Phases / Structured Query Execution Pipeline)

Attribute / Phase	Description					
analyzed	<p>Analyzed logical plan that has passed Analyzer's check rules.</p> <table border="1"> <tr> <td>Tip</td><td>Beside <code>analyzed</code>, you can use <code>Dataset.explain</code> basic action (with <code>extended</code> flag enabled) or SQL's <code>EXPLAIN EXTENDED</code> to see the analyzed logical plan of a structured query.</td></tr> </table>		Tip	Beside <code>analyzed</code> , you can use <code>Dataset.explain</code> basic action (with <code>extended</code> flag enabled) or SQL's <code>EXPLAIN EXTENDED</code> to see the analyzed logical plan of a structured query.		
Tip	Beside <code>analyzed</code> , you can use <code>Dataset.explain</code> basic action (with <code>extended</code> flag enabled) or SQL's <code>EXPLAIN EXTENDED</code> to see the analyzed logical plan of a structured query.					
withCachedData	<p>analyzed logical plan after <code>cacheManager</code> was requested to replace logical query segments with cached query plans.</p> <p><code>withCachedData</code> makes sure that the logical plan was analyzed and uses supported operations only.</p>					
optimizedPlan	<p>Optimized logical plan that is the result of executing the logical query plan optimizer on the withCachedData logical plan.</p>					
sparkPlan	<p>Physical plan (after SparkPlanner has planned the optimized logical plan).</p> <table border="1"> <tr> <td>Note</td><td><code>sparkPlan</code> is the first physical plan from the collection of all possible physical plans.</td></tr> <tr> <td>Note</td><td>It is guaranteed that Catalyst's <code>QueryPlanner</code> (which <code>SparkPlanner</code> extends) will always generate at least one physical plan.</td></tr> </table>		Note	<code>sparkPlan</code> is the first physical plan from the collection of all possible physical plans.	Note	It is guaranteed that Catalyst's <code>QueryPlanner</code> (which <code>SparkPlanner</code> extends) will always generate at least one physical plan .
Note	<code>sparkPlan</code> is the first physical plan from the collection of all possible physical plans.					
Note	It is guaranteed that Catalyst's <code>QueryPlanner</code> (which <code>SparkPlanner</code> extends) will always generate at least one physical plan .					
	<p>Optimized physical query plan that is in the final optimized "shape" and therefore ready for execution, i.e. the physical sparkPlan with physical preparation rules applied.</p> <table border="1"> <tr> <td>Note</td><td>Amongst the physical optimization rules that <code>executedPlan</code> phase triggers is the CollapseCodegenStages physical preparation rule that collapses physical operators that support code generation together as a WholeStageCodegenExec operator.</td></tr> </table>		Note	Amongst the physical optimization rules that <code>executedPlan</code> phase triggers is the CollapseCodegenStages physical preparation rule that collapses physical operators that support code generation together as a WholeStageCodegenExec operator.		
Note	Amongst the physical optimization rules that <code>executedPlan</code> phase triggers is the CollapseCodegenStages physical preparation rule that collapses physical operators that support code generation together as a WholeStageCodegenExec operator.					

<p><code>executedPlan</code></p>	<p><code>executedPlan</code> physical plan is used when:</p> <ul style="list-style-type: none"> • <code>Dataset.explain</code> operator is used to show the logical and physical query plans of a structured query • <code>Dataset.localCheckpoint</code> and <code>Dataset.checkpoint</code> operators are used (through <code>checkpoint</code>) • <code>Dataset.foreach</code> and <code>Dataset.foreachPartition</code> actions are used (through <code>withNewRDDExecutionId</code>) • <code>Dataset</code> is requested to <code>execute an action under a new execution ID</code> (e.g. for the <code>Dataset</code> operators: <code>collect</code>, <code>count</code>, <code>head</code> and <code>toLocalIterator</code>) • <code>CacheManager</code> is requested to <code>cacheQuery</code> (e.g. for <code>Dataset.persist</code> basic action) or <code>recacheByCondition</code> • <code>QueryExecution</code> is requested for the <code>RDD[InternalRow]</code> of a structured query (in the <code>toRdd</code> query execution phase), <code>simpleString</code>, <code>toString</code>, <code>stringWithStats</code>, <code>codegenToSeq</code>, and the <code>Hive-compatible output format</code> • <code>SQLExecution</code> is requested to execute a <code>Dataset action under a new execution id</code> • <code>PlanSubqueries</code> physical query optimization is <code>executed</code> • <code>AnalyzeColumnCommand</code> and <code>ExplainCommand</code> logical commands are <code>executed</code> • <code>DebugQuery</code> is requested for <code>debug</code> and <code>debugCodegen</code> 		
<p><code>RDD</code> of <code>internal binary rows</code> (i.e. <code>RDD[InternalRow]</code>) after executing the <code>executedPlan</code>.</p> <p>The <code>RDD</code> is the top-level <code>RDD</code> of the <code>DAG</code> of <code>RDDs</code> (that represent physical operators).</p>	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%; background-color: #f0f0f0; vertical-align: top; padding: 5px;"> <p>Note</p> </td><td style="vertical-align: top; padding: 5px;"> <p><code>toRdd</code> is a "boundary" between two Spark modules: Spark SQL and Spark Core.</p> </td></tr> </table> <p><code>Note</code> After you have executed <code>toRdd</code> (directly or not), you basically "leave" Spark SQL's Dataset world and "enter" Spark Core's RDD space.</p>	<p>Note</p>	<p><code>toRdd</code> is a "boundary" between two Spark modules: Spark SQL and Spark Core.</p>
<p>Note</p>	<p><code>toRdd</code> is a "boundary" between two Spark modules: Spark SQL and Spark Core.</p>		

`toRdd` triggers a structured query execution (i.e. physical planning, but not execution of the plan) using `SparkPlan.execute` that recursively triggers execution of every child physical operator in the physical plan tree.

Note	You can use <code>SparkSession.internalCreateDataFrame</code> to apply a schema to an <code>RDD[InternalRow]</code> .
------	---

Note	Use <code>Dataset.rdd</code> to access the <code>RDD[InternalRow]</code> with internal binary rows deserialized to a Scala type.
------	--

You can access the lazy attributes as follows:

```
val dataset: Dataset[Long] = ...
dataset.queryExecution.executedPlan
```

`QueryExecution` uses the [Catalyst Query Optimizer](#) and [Tungsten](#) for better structured query performance.

Table 2. `QueryExecution`'s Properties

Name	Description
<code>planner</code>	SparkPlanner

`QueryExecution` uses the input `SparkSession` to access the current [SparkPlanner](#) (through [SessionState](#)) when it is created. It then computes a [SparkPlan](#) (a `PhysicalPlan` exactly) using the planner. It is available as the `sparkPlan` attribute.

Note	A variant of <code>QueryExecution</code> that Spark Structured Streaming uses for query planning is <code>IncrementalExecution</code> . Refer to IncrementalExecution — QueryExecution of Streaming Datasets in the Spark Structured Streaming gitbook.
------	---

Tip	Use <code>explain</code> operator to know about the logical and physical plans of a <code>dataset</code> .
-----	--

```

val ds = spark.range(5)
scala> ds.queryExecution
res17: org.apache.spark.sql.execution.QueryExecution =
== Parsed Logical Plan ==
Range 0, 5, 1, 8, [id#39L]

== Analyzed Logical Plan ==
id: bigint
Range 0, 5, 1, 8, [id#39L]

== Optimized Logical Plan ==
Range 0, 5, 1, 8, [id#39L]

== Physical Plan ==
WholeStageCodegen
: +- Range 0, 1, 8, 5, [id#39L]

```

Note	<code>QueryExecution</code> belongs to <code>org.apache.spark.sql.execution</code> package.
------	---

Note	<code>QueryExecution</code> is a transient feature of a <code>Dataset</code> , i.e. it is not preserved across serializations.
------	--

Text Representation With Statistics — `stringWithStats` Method

	<code>stringWithStats: String</code>
--	--------------------------------------

	<code>stringWithStats ...FIXME</code>
--	---------------------------------------

Note	<code>stringWithStats</code> is used exclusively when <code>ExplainCommand</code> logical command is executed (with <code>cost</code> flag enabled).
------	--

debug Object

Caution	<code>FIXME</code>
---------	--------------------

Building Complete Text Representation — `completeString` Internal Method

Caution	<code>FIXME</code>
---------	--------------------

Creating QueryExecution Instance

`QueryExecution` takes the following when created:

- `SparkSession`
- `Logical plan`

Physical Query Optimizations (Physical Plan Preparation Rules) — `preparations` Method

```
preparations: Seq[Rule[SparkPlan]]
```

`preparations` is the set of the physical query optimization rules that transform a [physical query plan](#) to be more efficient and optimized for execution (i.e. `Rule[SparkPlan]`).

The `preparations` physical query optimizations are applied sequentially (one by one) to a physical plan in the following order:

1. [ExtractPythonUDFs](#)
2. [PlanSubqueries](#)
3. [EnsureRequirements](#)
4. [CollapseCodegenStages](#)
5. [ReuseExchange](#)
6. [ReuseSubquery](#)

Note

`preparations` rules are used when:

- `QueryExecution` is requested for the [executedPlan](#) physical plan (through `prepareForExecution`)
- (Spark Structured Streaming) `IncrementalExecution` is requested for the physical optimization rules for streaming structured queries

Applying preparations Physical Query Optimization Rules to Physical Plan — `prepareForExecution` Method

```
prepareForExecution(plan: SparkPlan): SparkPlan
```

`prepareForExecution` takes [physical preparation rules](#) and applies them one by one to the input `physical plan`.

Note

`prepareForExecution` is used exclusively when `QueryExecution` is requested to [prepare the physical plan for execution](#).

assertSupported Method

```
assertSupported(): Unit
```

`assertSupported` requests `UnsupportedOperationChecker` to [checkForBatch](#) when...FIXME

Note

`assertSupported` is used exclusively when `QueryExecution` is requested for [withCachedData](#) logical plan.

Creating Analyzed Logical Plan and Checking Correctness — assertAnalyzed Method

```
assertAnalyzed(): Unit
```

`assertAnalyzed` triggers initialization of `analyzed` (which is almost like executing it).

Note

`assertAnalyzed` executes `analyzed` by accessing it and throwing the result away. Since `analyzed` is a lazy value in Scala, it will then get initialized for the first time and stays so forever.

`assertAnalyzed` then requests `Analyzer` to [validate analysis of the logical plan](#) (i.e. `analyzed`).

Note

`assertAnalyzed` uses `SparkSession` to access the current `SessionState` that it then uses to [access the Analyzer](#) .

In Scala the access path looks as follows.

```
sparkSession.sessionState.analyzer
```

In case of any `AnalysisException` , `assertAnalyzed` creates a new `AnalysisException` to make sure that it holds `analyzed` and reports it.

Note

- `assertAnalyzed` is used when:
- `Dataset` is created
 - `QueryExecution` is requested for `LogicalPlan` with cached data
 - `CreateViewCommand` and `AlterViewAsCommand` are executed

Building Text Representation with Cost Stats — `toStringWithStats` Method

```
toStringWithStats: String
```

`toStringWithStats` is a mere alias for `completeString` with `appendStats` flag enabled.

Note

`toStringWithStats` is a custom `toString` with `cost` statistics.

```
// test dataset
val dataset = spark.range(20).limit(2)

// toStringWithStats in action - note Optimized Logical Plan section with Statistics
scala> dataset.queryExecution.toStringWithStats
res6: String =
== Parsed Logical Plan ==
GlobalLimit 2
+- LocalLimit 2
  +- Range (0, 20, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
GlobalLimit 2
+- LocalLimit 2
  +- Range (0, 20, step=1, splits=Some(8))

== Optimized Logical Plan ==
GlobalLimit 2, Statistics(sizeInBytes=32.0 B, rowCount=2, isBroadcastable=false)
+- LocalLimit 2, Statistics(sizeInBytes=160.0 B, isBroadcastable=false)
  +- Range (0, 20, step=1, splits=Some(8)), Statistics(sizeInBytes=160.0 B, isBroadcastable=false)

== Physical Plan ==
CollectLimit 2
+- *Range (0, 20, step=1, splits=Some(8))
```

Note

`toStringWithStats` is used exclusively when `ExplainCommand` is executed (only when `cost` attribute is enabled).

Transforming SparkPlan Execution Result to Hive-Compatible Output Format — `hiveResultString` Method

```
hiveResultString(): Seq[String]
```

`hiveResultString` returns the result as a Hive-compatible output format.

```
scala> spark.range(5).queryExecution.hiveResultString
res0: Seq[String] = ArrayBuffer(0, 1, 2, 3, 4)

scala> spark.read.csv("people.csv").queryExecution.hiveResultString
res4: Seq[String] = ArrayBuffer(id      name    age, 0  Jacek  42)
```

Internally, `hiveResultString` transformation the [SparkPlan](#).

Table 3. `hiveResultString`'s SparkPlan Transformations (in execution order)

SparkPlan	Description
ExecutedCommandExec for DescribeTableCommand	Executes <code>DescribeTableCommand</code> and transforms every Row to a Hive-compatible output format.
ExecutedCommandExec for ShowTablesCommand	Executes <code>ExecutedCommandExec</code> and transforms the result to a collection of table names.
Any other SparkPlan	Executes <code>SparkPlan</code> and transforms the result to a Hive-compatible output format.

Note	<code>hiveResultString</code> is used exclusively when <code>SparksQLDriver</code> (of <code>ThriftServer</code>) runs a command.
------	--

Extended Text Representation with Logical and Physical Plans — `toString` Method

```
toString: String
```

Note	<code>toString</code> is part of Java's <code>Object</code> Contract to...FIXME.
------	--

`toString` is a mere alias for [completeString](#) with `appendStats` flag disabled.

Note	<code>toString</code> is on the "other" side of toStringWithStats which has <code>appendStats</code> flag enabled.
------	--

Simple (Basic) Text Representation — `simpleString` Method

```
simpleString: String
```

`simpleString` requests the [optimized SparkPlan](#) for the [text representation](#) (of all nodes in the query tree) with `verbose` flag turned off.

In the end, `simpleString` adds == Physical Plan == header to the text representation and [redacts sensitive information](#).

```
import org.apache.spark.sql.{functions => f}
val q = spark.range(10).withColumn("rand", f.rand())
val output = q.queryExecution.simpleString

scala> println(output)
== Physical Plan ==
*(1) Project [id#5L, rand(6017561978775952851) AS rand#7]
+- *(1) Range (0, 10, step=1, splits=8)
```

Note

`simpleString` is used when:

- `ExplainCommand` is [executed](#)
- Spark Structured Streaming's `StreamingExplainCommand` is [executed](#)

Redacting Sensitive Information — `withRedaction` Internal Method

```
withRedaction(message: String): String
```

`withRedaction` takes the value of [spark.sql.redaction.string.regex](#) configuration property (as the regular expression to point at sensitive information) and requests Spark Core's `utils` to redact sensitive information in the input `message`.

Note

Internally, Spark Core's `utils.redact` uses Java's `Regex.replaceAllIn` to replace all matches of a pattern with a string.

Note

`withRedaction` is used when `QueryExecution` is requested for the [simple](#), [extended](#) and [with statistics](#) text representations.

UnsupportedOperationChecker

UnsupportedOperationChecker is...FIXME

checkForBatch Method

```
checkForBatch(plan: LogicalPlan): Unit
```

checkForBatch ...FIXME

Note

checkForBatch is used when...FIXME

Analyzer — Logical Query Plan Analyzer

`Analyzer` (aka *Spark Analyzer* or *Query Analyzer*) is the **logical query plan analyzer** that **semantically validates and transforms an unresolved logical plan** to an **analyzed logical plan**.

`Analyzer` is a concrete `RuleExecutor` of `LogicalPlan` (i.e. `RuleExecutor[LogicalPlan]`) with the **logical evaluation rules**.

```
Analyzer: Unresolved Logical Plan ==> Analyzed Logical Plan
```

`Analyzer` uses `SessionCatalog` while resolving relational entities, e.g. databases, tables, columns.

`Analyzer` is created when `sessionState` is requested for the analyzer.

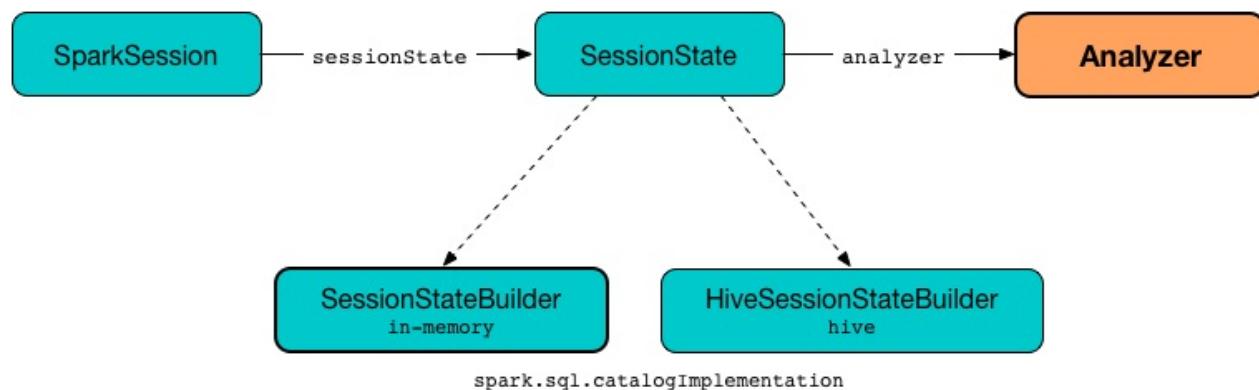


Figure 1. Creating Analyzer

`Analyzer` is available as the `analyzer` property of a session-specific `SessionState`.

```

scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.analyzer
org.apache.spark.sql.catalyst.analysis.Analyzer

```

You can access the analyzed logical plan of a structured query (as a `Dataset`) using `Dataset.explain` basic action (with `extended` flag enabled) or SQL's `EXPLAIN EXTENDED` SQL command.

```
// sample structured query
val inventory = spark
  .range(5)
  .withColumn("new_column", 'id + 5 as "plus5")

// Using explain operator (with extended flag enabled)
scala> inventory.explain(extended = true)
== Parsed Logical Plan ==
Project [id#0L, ('id + 5) AS plus5#2 AS new_column#3]
+- AnalysisBarrier
  +- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint, new_column: bigint
Project [id#0L, (id#0L + cast(5 as bigint)) AS new_column#3L]
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#0L, (id#0L + 5) AS new_column#3L]
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
*(1) Project [id#0L, (id#0L + 5) AS new_column#3L]
+- *(1) Range (0, 5, step=1, splits=8)
```

Alternatively, you can access the analyzed logical plan using `QueryExecution` and its `analyzed` property (that together with `numberedTreeString` method is a very good "debugging" tool).

```
val analyzedPlan = inventory.queryExecution.analyzed
scala> println(analyzedPlan.numberedTreeString)
00 Project [id#0L, (id#0L + cast(5 as bigint)) AS new_column#3L]
01 +- Range (0, 5, step=1, splits=Some(8))
```

`Analyzer` defines `extendedResolutionRules` extension point for additional logical evaluation rules that a custom `Analyzer` can use to extend the `Resolution` rule batch. The rules are added at the end of the `Resolution` batch.

Note

`SessionState` uses its own `Analyzer` with custom `extendedResolutionRules`, `postHocResolutionRules`, and `extendedCheckRules` extension methods.

Table 1. Analyzer's Internal Registries and Counters

Name	Description
extendedResolutionRules	Additional rules for Resolution batch. Empty by default
fixedPoint	FixedPoint with maxIterations for Hints , Substitution , Resolution and Cleanup batches. Set when Analyzer is created (and can be defined explicitly or through optimizerMaxIterations configuration setting).
postHocResolutionRules	The only rules in Post-Hoc Resolution batch if defined (that are executed in one pass, i.e. <code>once</code> strategy). Empty by default

`Analyzer` is used by `QueryExecution` to [resolve the managed `LogicalPlan`](#) (and, as a sort of follow-up, [assert that a structured query has already been properly analyzed](#), i.e. no failed or unresolved or somehow broken logical plan operators and expressions exist).

Enable `TRACE` or `DEBUG` logging levels for the respective session-specific loggers to see what happens inside `Analyzer`.

- `org.apache.spark.sql.internal.SessionState$$anon$1`
- `org.apache.spark.sql.hive.HiveSessionStateBuilder$$anon$1` when [Hive support is enabled](#)

Add the following line to `conf/log4j.properties`:

```
# with no Hive support
log4j.logger.org.apache.spark.sql.internal.SessionState$$anon$1=TRACE

# with Hive support enabled
log4j.logger.org.apache.spark.sql.hive.HiveSessionStateBuilder$$anon$1=DEBUG
```

Refer to [Logging](#).

The reason for such weird-looking logger names is that `analyzer` attribute is created as an anonymous subclass of `Analyzer` class in the respective `SessionStates`.

Executing Logical Evaluation Rules — `execute` Method

`Analyzer` is a [RuleExecutor](#) that defines the [logical rules](#) (i.e. resolving, removing, and in general modifying it), e.g.

- Resolves unresolved [relations](#) and [functions](#) (including [UnresolvedGenerators](#)) using provided [SessionCatalog](#)
- ...

Table 2. Analyzer's Batches

Batch Name	Strategy	Rules	
Hints	FixedPoint	ResolveBroadcastHints	Resolves U
		ResolveCoalesceHints	Resolves U
		RemoveAllHints	Removes a
Simple Sanity Check	Once	LookupFunctions	Checks wh NoSuchFunc
Substitution	FixedPoint	CTESubstitution	Resolves V
		WindowsSubstitution	Substitutes
		EliminateUnions	Eliminates
		SubstituteUnresolvedOrdinals	Replaces o
		ResolveTableValuedFunctions	Replaces T
		ResolveRelations	Resolves: <ul style="list-style-type: none"> • InsertInto • Unresolv
		ResolveReferences	
		ResolveCreateNamedStruct	Resolves C
		ResolveDeserializer	
		ResolveNewInstance	
		ResolveUpCast	
			Resolves g <ul style="list-style-type: none"> • Cube , • Filter

			<ul style="list-style-type: none"> Sort <p>Expectsthat some iteration</p> <p>Fails analysis</p> <pre>scala> sq org.apache.spark.sql.catalyst.expressions.GeneratedClass\$CodegenFunctionBuilder\$</pre>
		ResolveGroupingAnalytics	
		ResolvePivot	<p>Resolves P a single Ag</p>
		ResolveOrdinalInOrderByAndGroupBy	
		ResolveMissingReferences	
Resolution	FixedPoint	ExtractGenerator	
		ResolveGenerate	
		ResolveFunctions	<p>Resolves functions</p> <ul style="list-style-type: none"> Unresolved function Unresolved reference <p>If generator is present:</p> <p>[name] generator</p>
		ResolveAliases	<p>Replaces aliases</p> <ul style="list-style-type: none"> Name conflicts Multiple aliases Alias conflicts
		ResolveSubquery	Resolves subqueries
		ResolveWindowOrder	

	ResolveWindowFrame	Resolves W	
	ResolveNaturalAndUsingJoin		
	ExtractWindowExpressions		
	GlobalAggregates	Resolves (a logical oper	
	ResolveAggregateFunctions	Resolves a Note F	
	TimeWindowing	Resolves T	
	ResolveInlineTables	Resolves U	
	TypeCoercion.typeCoercionRules	Type coerci	
	extendedResolutionRules		
Post-Hoc Resolution	Once	postHocResolutionRules	
View	Once	AliasViewChild	
Nondeterministic	Once	PullOutNondeterministic	
UDF	Once	HandleNullInputsForUDF	
FixNullability	Once	FixNullability	
ResolveTimeZone	Once	ResolveTimeZone	Replaces T
Cleanup	FixedPoint	CleanupAliases	

Tip Consult the sources of [Analyzer](#) for the up-to-date list of the evaluation rules.

Creating Analyzer Instance

`Analyzer` takes the following when created:

- `SessionCatalog`

- [CatalystConf](#)
- Maximum number of iterations (of the [FixedPoint](#) rule batches, i.e. [Hints](#), [Substitution](#), [Resolution](#) and [Cleanup](#))

`Analyzer` initializes the [internal registries and counters](#).

Note

`Analyzer` can also be created without specifying the `maxIterations` argument which is then configured using `optimizerMaxIterations` configuration setting.

resolver Method

```
resolver: Resolver
```

`resolver` requests [CatalystConf](#) for [Resolver](#).

Note

`Resolver` is a mere function of two `String` parameters that returns `true` if both refer to the same entity (i.e. for case insensitive equality).

resolveExpression Method

```
resolveExpression(  
    expr: Expression,  
    plan: LogicalPlan,  
    throws: Boolean = false): Expression
```

`resolveExpression` ...FIXME

Note

`resolveExpression` is a `protected[sql]` method.

Note

`resolveExpression` is used when...FIXME

commonNaturalJoinProcessing Internal Method

```
commonNaturalJoinProcessing(  
    left: LogicalPlan,  
    right: LogicalPlan,  
    joinType: JoinType,  
    joinNames: Seq[String],  
    condition: Option[Expression]): Project
```

`commonNaturalJoinProcessing` ...FIXME

Note	commonNaturalJoinProcessing is used when...FIXME
------	--

executeAndCheck Method

```
executeAndCheck(plan: LogicalPlan): LogicalPlan
```

executeAndCheck ...FIXME

Note	executeAndCheck is used exclusively when <code>QueryExecution</code> is requested for the analyzed logical plan.
------	--

CheckAnalysis — Analysis Validation

`CheckAnalysis` defines `checkAnalysis` method that `Analyzer` uses to check if a `logical plan` is correct (after all the transformations) by applying `validation rules` and in the end marking it as analyzed.

Note	An analyzed logical plan is correct and ready for execution.
------	--

`CheckAnalysis` defines `extendedCheckRules` extension point that allows for extra analysis check rules.

Validating Analysis of Logical Plan (and Marking Plan As Analyzed) — `checkAnalysis` Method

```
checkAnalysis(plan: LogicalPlan): Unit
```

`checkAnalysis` recursively checks the correctness of the analysis of the input `logical plan` and [marks it as analyzed](#).

Note	<code>checkAnalysis</code> fails analysis when finds <code>UnresolvedRelation</code> in the input <code>LogicalPlan</code> ... FIXME What else?
------	--

Internally, `checkAnalysis` processes nodes in the input `plan` (starting from the leafs, i.e. nodes down the operator tree).

`checkAnalysis` skips [logical plans that have already undergone analysis](#).

Table 1. `checkAnalysis`

LogicalPlan/Operator	
<code>UnresolvedRelation</code>	Fails analysis with the error message: <div style="background-color: #f0f0f0; padding: 5px;">Table or view not found: [tableIdentifier]</div>
<code>Unresolved Attribute</code>	Fails analysis with the error message: <div style="background-color: #f0f0f0; padding: 5px;">cannot resolve '[expr]' given input columns: [from]</div>
<code>Expression with incorrect input data types</code>	Fails analysis with the error message: <div style="background-color: #f0f0f0; padding: 5px;">cannot resolve '[expr]' due to data type mismatch: [message]</div>

<code>WindowExpressions</code> with a window function that is not one of the following expressions: <code>AggregateExpression</code> , <code>AggregateWindowFunction</code> or <code>OffsetWindowFunction</code>	Fails analysis with the error message: Expression '[e]' not supported within a window function.
<code>Nondeterministic expressions</code>	FIXME
<code>UnresolvedHint</code>	FIXME
<code>FIXME</code>	FIXME

After the validations, `checkAnalysis` executes additional check rules for correct analysis.

`checkAnalysis` then checks if `plan` is analyzed correctly (i.e. no logical plans are left unresolved). If there is one, `checkAnalysis` fails the analysis with `AnalysisException` and the following error message:

```
unresolved operator [o.simpleString]
```

In the end, `checkAnalysis` marks the entire logical plan as analyzed.

Note	<code>checkAnalysis</code> is used when: <ul style="list-style-type: none"> <code>QueryExecution</code> creates analyzed logical plan and checks its correctness (which happens mostly when a <code>Dataset</code> is created) <code>ExpressionEncoder</code> does <code>resolveAndBind</code> <code>ResolveAggregateFunctions</code> is executed (for <code>Sort</code> logical plan)
------	---

Extended Analysis Check Rules — `extendedCheckRules` Extension Point

```
extendedCheckRules: Seq[LogicalPlan => Unit]
```

`extendedCheckRules` is a collection of rules (functions) that `checkAnalysis` uses for custom analysis checks (after the main validations have been executed).

Note	When a condition of a rule does not hold the function throws an <code>AnalysisException</code> directly or using <code>failAnalysis</code> method.
------	--

checkSubqueryExpression Internal Method

```
checkSubqueryExpression(plan: LogicalPlan, expr: SubqueryExpression): Unit
```

checkSubqueryExpression ...FIXME

Note

checkSubqueryExpression is used exclusively when CheckAnalysis is requested to validate analysis of a logical plan (for SubqueryExpression expressions).

SparkOptimizer — Logical Query Plan Optimizer

`SparkOptimizer` is a concrete [logical query plan optimizer](#) with additional [optimization rules](#) (that extend the [base logical optimization rules](#)).

`SparkOptimizer` gives three extension points for additional optimization rules:

1. Pre-Optimization Batches
2. Post-Hoc Optimization Batches
3. User Provided Optimizers (as [extraOptimizations](#) of the [ExperimentalMethods](#))

`SparkOptimizer` is [created](#) when `SessionState` is requested for the [Logical Optimizer](#) the first time (through [BaseSessionStateBuilder](#)).

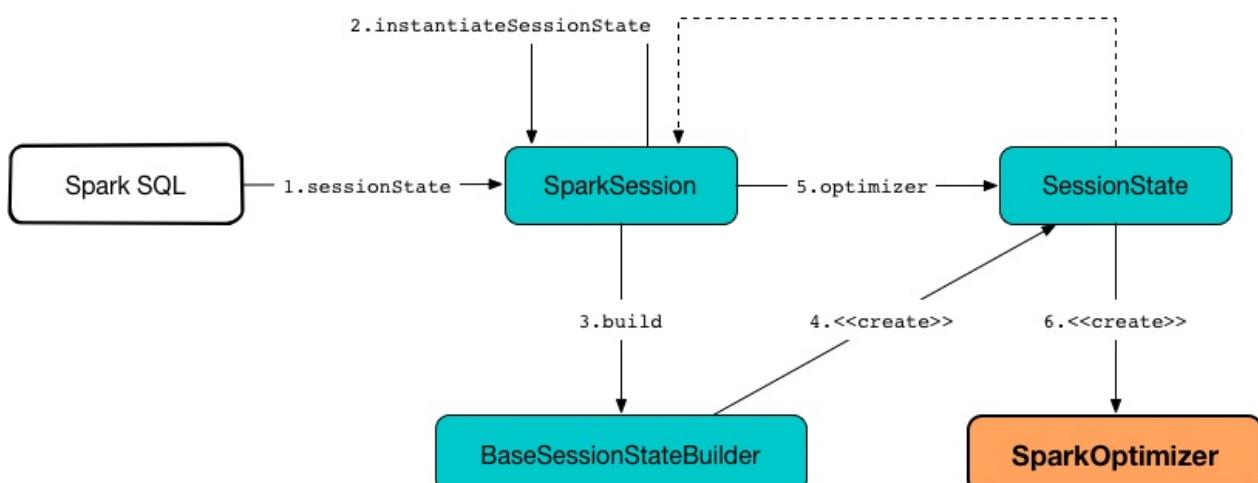


Figure 1. Creating SparkOptimizer

`SparkOptimizer` is available as the [optimizer](#) property of a session-specific `SessionState`.

```

scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.optimizer
org.apache.spark.sql.catalyst.optimizer.Optimizer

// It is a SparkOptimizer really.
// Let's check that out with a type cast

import org.apache.spark.sql.execution.SparkOptimizer
scala> spark.sessionState.optimizer.isInstanceOf[SparkOptimizer]
res1: Boolean = true
  
```

You can access the optimization logical plan of a structured query through the `QueryExecution as optimizedPlan`.

```
// Applying two filter in sequence on purpose
// We want to kick CombineTypedFilters optimizer in
val dataset = spark.range(10).filter(_ % 2 == 0).filter(_ == 0)

// optimizedPlan is a lazy value
// Only at the first time you call it you will trigger optimizations
// Next calls end up with the cached already-optimized result
// Use explain to trigger optimizations again
scala> dataset.queryExecution.optimizedPlan
res0: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], new
Instance(class java.lang.Long)
+- Range (0, 10, step=1, splits=Some(8))
```

SparkOptimizer defines the custom [default rule batches](#).

Table 1. SparkOptimizer's Default Optimization Batch Rules (in the order of execution)

Batch Name	Strategy	Rules	Description
		preOptimizationBatches	
		Base Logical Optimization Batches	
Optimize Metadata Only Query	Once	OptimizeMetadataOnlyQuery	
Extract Python UDF from Aggregate	Once	ExtractPythonUDFFromAggregate	
Prune File Source Table Partitions	Once	PruneFileSourcePartitions	
Push down operators to data source scan	Once	PushDownOperatorsToDataSource	Pushes down operators to underlying data sources (i.e. <code>DataSourceV2Relation</code>)
		postHocOptimizationBatches	
User Provided Optimizers	FixedPoint	extraOptimizations of the ExperimentalMethods	

SparkOptimizer considers `ExtractPythonUDFFromAggregate` optimization rule as [non-excludable](#).

Tip

Enable `DEBUG` or `TRACE` logging levels for `org.apache.spark.sql.execution.SparkOptimizer` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.SparkOptimizer=TRACE
```

Refer to [Logging](#).

Creating SparkOptimizer Instance

`SparkOptimizer` takes the following when created:

- `SessionCatalog`
- `ExperimentalMethods`

Extension Point for Additional Pre-Optimization Batches — `preOptimizationBatches` Method

```
preOptimizationBatches: Seq[Batch]
```

`preOptimizationBatches` are the additional **pre-optimization batches** that are executed right before the [regular optimization batches](#).

Extension Point for Additional Post-Hoc Optimization Batches — `postHocOptimizationBatches` Method

```
postHocOptimizationBatches: Seq[Batch] = Nil
```

`postHocOptimizationBatches` are the additional **post-optimization batches** that are executed right after the [regular optimization batches](#) (before [User Provided Optimizers](#)).

Further Reading and Watching

1. [Deep Dive into Spark SQL's Catalyst Optimizer](#)
2. (video) [Modern Spark DataFrame and Dataset \(Intermediate Tutorial\)](#) by Adam Breindel

Catalyst Optimizer — Generic Logical Query Plan Optimizer

`optimizer` (aka **Catalyst Optimizer**) is the base of [logical query plan optimizers](#) that defines the [rule batches of logical optimizations](#) (i.e. logical optimizations that are the rules that transform the query plan of a structured query to produce the [optimized logical plan](#)).

Note

`SparkOptimizer` is the one and only direct implementation of the `optimizer` Contract in Spark SQL.

`optimizer` is a [RuleExecutor](#) of [LogicalPlan](#) (i.e. `RuleExecutor[LogicalPlan]`).

Optimizer: Analyzed Logical Plan ==> Optimized Logical Plan

`optimizer` is available as the [optimizer](#) property of a session-specific [SessionState](#) .

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.optimizer
org.apache.spark.sql.catalyst.optimizer.Optimizer
```

You can access the optimized logical plan of a structured query (as a [Dataset](#)) using [Dataset.explain](#) basic action (with `extended` flag enabled) or SQL's `EXPLAIN EXTENDED` SQL command.

```
// sample structured query
val inventory = spark
  .range(5)
  .withColumn("new_column", 'id + 5 as "plus5")

// Using explain operator (with extended flag enabled)
scala> inventory.explain(extended = true)
== Parsed Logical Plan ==
'Project [id#0L, ('id + 5) AS plus5#2 AS new_column#3]
+- AnalysisBarrier
  +- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint, new_column: bigint
Project [id#0L, (id#0L + cast(5 as bigint)) AS new_column#3L]
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#0L, (id#0L + 5) AS new_column#3L]
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
*(1) Project [id#0L, (id#0L + 5) AS new_column#3L]
+- *(1) Range (0, 5, step=1, splits=8)
```

Alternatively, you can access the analyzed logical plan using `QueryExecution` and its `optimizedPlan` property (that together with `numberedTreeString` method is a very good "debugging" tool).

```
val optimizedPlan = inventory.queryExecution.optimizedPlan
scala> println(optimizedPlan.numberedTreeString)
00 Project [id#0L, (id#0L + 5) AS new_column#3L]
01 +- Range (0, 5, step=1, splits=Some(8))
```

`Optimizer` defines the `default rule batches` that are considered the base rule batches that can be further refined (extended or `with some rules excluded`).

Table 1. Optimizer's Default Optimization Rule Batches (in the order of execution)

Batch Name	Strategy	Rules
Eliminate Distinct	Once	EliminateDistinct
		EliminateSubqueryAliases

		EliminateView
Finish Analysis	Once	ReplaceExpressions
		ComputeCurrentTime
		GetCurrentDatabase
		RewriteDistinctAggregates
		ReplaceDeduplicateWithAggregate
Union	Once	CombineUnions
LocalRelation early	FixedPoint	ConvertToLocalRelation
		PropagateEmptyRelation
Pullup Correlated Expressions	Once	PullupCorrelatedPredicates
Subquery	Once	OptimizeSubqueries
	FixedPoint	RewriteExceptAll
		RewriteIntersectAll
		ReplaceIntersectWithSemiJoin
		ReplaceExceptWithFilter
		ReplaceExceptWithAntiJoin
Replace Operators		ReplaceDistinctWithAggregate
		RemoveLiteralFromGroupExpressions

Aggregate	FixedPoint	RemoveRepetitionFromGroupExpressions
operatorOptimizationBatch		
Join Reorder	Once	CostBasedJoinReorder
Remove Redundant Sorts	Once	RemoveRedundantSorts
Decimal Optimizations	FixedPoint	DecimalAggregates
Object Expressions Optimization	FixedPoint	EliminateMapObjects
		CombineTypedFilters
LocalRelation	FixedPoint	ConvertToLocalRelation
		PropagateEmptyRelation
Extract PythonUDF From JoinCondition	Once	PullOutPythonUDFInJoinCondition
Check Cartesian Products	Once	CheckCartesianProducts
RewriteSubquery	Once	RewritePredicateSubquery
		ColumnPruning
		CollapseProject
		RemoveRedundantProject
UpdateAttributeReferences	Once	UpdateNullabilityInAttributeReferences

Tip

Consult the sources of the `Optimizer` class for the up-to-date list of the `default optimization rule batches`.

`Optimizer` defines the `operator optimization rules` with the `extendedOperatorOptimizationRules` extension point for additional optimizations in the **Operator Optimization** batch.

Table 2. Optimizer's Operator Optimization Rules (in the order of execution)

Rule Name	Description

PushProjectionThroughUnion	
ReorderJoin	
EliminateOuterJoin	
PushPredicateThroughJoin	
PushDownPredicate	
LimitPushDown	
ColumnPruning	
CollapseRepartition	
CollapseProject	
CollapseWindow	Collapses two adjacent Window logical operators
CombineFilters	
CombineLimits	
CombineUnions	
NullPropagation	
ConstantPropagation	
FoldablePropagation	
Optimizeln	
ConstantFolding	
ReorderAssociativeOperator	
LikeSimplification	
BooleanSimplification	
SimplifyConditionals	

RemoveDisposableExpressions	
SimplifyBinaryComparison	
PruneFilters	
EliminateSorts	
SimplifyCasts	
SimplifyCaseConversionExpressions	
RewriteCorrelatedScalarSubquery	
EliminateSerialization	
RemoveRedundantAliases	
RemoveRedundantProject	
SimplifyExtractValueOps	
CombineConcats	

`optimizer` defines [Operator Optimization Batch](#) that is simply a collection of rule batches with the [operator optimization rules](#) before and after `InferFiltersFromConstraints` logical rule.

Table 3. Optimizer's Operator Optimization Batch (in the order of execution)

Batch Name	Strategy	Rules
Operator Optimization before Inferring Filters	FixedPoint	Operator optimization rules
Infer Filters	Once	<code>InferFiltersFromConstraints</code>
Operator Optimization after Inferring Filters	FixedPoint	Operator optimization rules

`optimizer` uses `spark.sql.optimizer.excludedRules` configuration property to control what optimization rules in the `defaultBatches` should be excluded (default: none).

`Optimizer` takes a [SessionCatalog](#) when created.

Note

`optimizer` is a Scala abstract class and cannot be created directly. It is created indirectly when the [concrete Optimizers](#) are.

`Optimizer` defines the [non-excludable optimization rules](#) that are considered critical for query optimization and will never be excluded (even if they are specified in `spark.sql.optimizer.excludedRules` configuration property).

Table 4. Optimizer’s Non-Excludable Optimization Rules

Rule Name	Description
PushProjectionThroughUnion	
EliminateDistinct	
EliminateSubqueryAliases	
EliminateView	
ReplaceExpressions	
ComputeCurrentTime	
GetCurrentDatabase	
RewriteDistinctAggregates	
ReplaceDeduplicateWithAggregate	
ReplaceIntersectWithSemiJoin	
ReplaceExceptWithFilter	
ReplaceExceptWithAntiJoin	
RewriteExceptAll	
RewriteIntersectAll	
ReplaceDistinctWithAggregate	
PullupCorrelatedPredicates	
RewriteCorrelatedScalarSubquery	
RewritePredicateSubquery	
PullOutPythonUDFInJoinCondition	

Table 5. Optimizer's Internal Registries and Counters

Name	Initial Value	Description
fixedPoint	FixedPoint with the number of iterations as defined by <code>spark.sql.optimizer.maxIterations</code>	Used in Replace Operators , Aggregate , Operator Optimizations , Decimal Optimizations , Typed Filter Optimization and LocalRelation batches (and also indirectly in the User Provided Optimizers rule batch in SparkOptimizer).

Additional Operator Optimization Rules — `extendedOperatorOptimizationRules` Extension Point

```
extendedOperatorOptimizationRules: Seq[Rule[LogicalPlan]]
```

`extendedOperatorOptimizationRules` extension point defines additional rules for the Operator Optimization batch.

Note

`extendedOperatorOptimizationRules` rules are executed right after [Operator Optimization before Inferring Filters](#) and [Operator Optimization after Inferring Filters](#).

batches Final Method

```
batches: Seq[Batch]
```

Note

`batches` is part of the [RuleExecutor Contract](#) to define the rule batches to use when executed.

`batches` ...FIXME

SparkPlanner — Spark Query Planner

`SparkPlanner` is a concrete [Catalyst Query Planner](#) that converts a [logical plan](#) to one or more [physical plans](#) using [execution planning strategies](#) with support for [extra strategies](#) (by means of [ExperimentalMethods](#)) and [extraPlanningStrategies](#).

Note	<code>SparkPlanner</code> is expected to plan (aka <i>generate</i>) at least one physical plan per logical plan .
------	--

`SparkPlanner` is available as [planner](#) of a `SessionState`.

```
val spark: SparkSession = ...
scala> :type spark.sessionState.planner
org.apache.spark.sql.execution.SparkPlanner
```

Table 1. SparkPlanner's Execution Planning Strategies (in execution order)

SparkStrategy	Description
<code>ExperimentalMethods</code> 's extraStrategies	
extraPlanningStrategies	Extension point for extra planning strategies
DataSourceV2Strategy	
FileSourceStrategy	
DataSourceStrategy	
SpecialLimits	
Aggregation	
JoinSelection	
InMemoryScans	
BasicOperators	

Note	<code>SparkPlanner</code> extends SparkStrategies abstract class.
------	---

Creating SparkPlanner Instance

`SparkPlanner` takes the following when created:

- `SparkContext`
- `SQLConf`
- `ExperimentalMethods`

Note

`SparkPlanner` is created in:

- `BaseSessionStateBuilder`
- `HiveSessionStateBuilder`
- Structured Streaming's `IncrementalExecution`

Extension Point for Extra Planning Strategies

— `extraPlanningStrategies` Method

```
extraPlanningStrategies: Seq[Strategy] = Nil
```

`extraPlanningStrategies` is an extension point to register extra [planning strategies](#) with the query planner.

Note

`extraPlanningStrategies` are executed after [extraStrategies](#).

Note

`extraPlanningStrategies` is used when `SparkPlanner` is requested for planning [strategies](#).

`extraPlanningStrategies` is overridden in the `SessionState` builders — [BaseSessionStateBuilder](#) and [HiveSessionStateBuilder](#).

Collecting PlanLater Physical Operators

— `collectPlaceholders` Method

```
collectPlaceholders(plan: SparkPlan): Seq[(SparkPlan, LogicalPlan)]
```

`collectPlaceholders` collects all [PlanLater](#) physical operators in the `plan` [physical plan](#).

Note

`collectPlaceholders` is part of [QueryPlanner Contract](#).

Pruning "Bad" Physical Plans — `prunePlans` Method

```
prunePlans(plans: Iterator[SparkPlan]): Iterator[SparkPlan]
```

`prunePlans` gives the input `plans` **physical plans** back (i.e. with no changes).

Note	<code>prunePlans</code> is part of QueryPlanner Contract to remove somehow "bad" plans.
------	---

Creating Physical Operator (Possibly Under FilterExec and ProjectExec Operators) — `pruneFilterProject` Method

```
pruneFilterProject(
  projectList: Seq[NamedExpression],
  filterPredicates: Seq[Expression],
  prunePushedDownFilters: Seq[Expression] => Seq[Expression],
  scanBuilder: Seq[Attribute] => SparkPlan): SparkPlan
```

Note	<code>pruneFilterProject</code> is almost like DataSourceStrategy.pruneFilterProjectRaw .
------	---

`pruneFilterProject` branches off per whether it is possible to use a column pruning only (to get the right projection) and the input `projectList` columns of this projection are enough to evaluate all input `filterPredicates` filter conditions.

If so, `pruneFilterProject` does the following:

1. Applies the input `scanBuilder` function to the input `projectList` columns that creates a new **physical operator**
2. If there are Catalyst predicate expressions in the input `prunePushedDownFilters` that cannot be pushed down, `pruneFilterProject` creates a **FilterExec** unary physical operator (with the unhandled predicate expressions)
3. Otherwise, `pruneFilterProject` simply returns the physical operator

Note	In this case no extra ProjectExec unary physical operator is created.
------	--

If not (i.e. it is neither possible to use a column pruning only nor evaluate filter conditions), `pruneFilterProject` does the following:

1. Applies the input `scanBuilder` function to the projection and filtering columns that creates a new **physical operator**
2. Creates a **FilterExec** unary physical operator (with the unhandled predicate expressions if available)

3. Creates a [ProjectExec](#) unary physical operator with the optional [FilterExec](#) operator (with the scan physical operator) or simply the scan physical operator alone

Note	<p><code>pruneFilterProject</code> is used when:</p> <ul style="list-style-type: none">• <code>HiveTableScans</code> execution planning strategy is executed (to plan HiveTableRelation leaf logical operators)• <code>InMemoryScans</code> execution planning strategy is executed (to plan InMemoryRelation leaf logical operators)
------	--

SparkStrategy — Base for Execution Planning Strategies

`SparkStrategy` is a Catalyst `GenericStrategy` that converts a `logical plan` into zero or more `physical plans`.

`SparkStrategy` marks `logical plans` (i.e. `LogicalPlan`) to be planned later (by some other `SparkStrategy` or after other `SparkStrategy` strategies have finished) using `PlanLater` physical operator.

```
planLater(plan: LogicalPlan): SparkPlan = PlanLater(plan)
```

	<p><code>SparkStrategy</code> is used as <code>Strategy</code> type alias (aka <i>type synonym</i>) in Spark's code base that is defined in <code>org.apache.spark.sql</code> package object, i.e.</p>
Note	<pre>type Strategy = SparkStrategy</pre>

PlanLater Physical Operator

Caution	FIXME
---------	-------

SparkStrategies — Container of Execution Planning Strategies

`SparkStrategies` is an abstract Catalyst [query planner](#) that *merely* serves as a "container" (or a namespace) of the concrete [execution planning strategies](#) (for [SparkPlanner](#)):

- [Aggregation](#)
- [BasicOperators](#)
- `FlatMapGroupsWithStateStrategy`
- [InMemoryScans](#)
- [JoinSelection](#)
- `SpecialLimits`
- `StatefulAggregationStrategy`
- `StreamingDeduplicationStrategy`
- `StreamingRelationStrategy`

`SparkStrategies` has a single lazily-instantiated `singleRowRdd` value that is an `RDD` of [internal binary rows](#) that [BasicOperators](#) execution planning strategy uses when resolving [OneRowRelation](#) (to [RDDScanExec](#) leaf physical operator).

Note	<code>OneRowRelation</code> logical operator represents SQL's SELECT clause without FROM clause or EXPLAIN DESCRIBE TABLE .
------	---

LogicalPlanStats — Statistics Estimates and Query Hints of Logical Operator

`LogicalPlanStats` adds statistics support to [logical operators](#) and is used for [query planning](#) (with or without [cost-based optimization](#), e.g. `CostBasedJoinReorder` or `JoinSelection`, respectively).

With `LogicalPlanStats` every logical operator has [statistics](#) that are computed only once when requested and are cached until [invalidated](#) and requested again.

Depending on [cost-based optimization](#) being enabled or not, `stats` computes the [statistics](#) with `FIXME` or `FIXED`, respectively.

Note

Cost-based optimization is enabled when `spark.sql.cbo.enabled` configuration property is turned on, i.e. `true`, and is disabled by default.

Use `EXPLAIN COST` SQL command to explain a query with the [statistics](#).

```
scala> sql("EXPLAIN COST SHOW TABLES").as[String].collect.foreach(println)
== Optimized Logical Plan ==
ShowTablesCommand false, Statistics(sizeInBytes=1.0 B, hints=None)

== Physical Plan ==
Execute ShowTablesCommand
+- ShowTablesCommand false
```

You can also access the statistics of a logical plan directly using `stats` method or indirectly requesting `QueryExecution` for [text representation with statistics](#).

```
val q = sql("SHOW TABLES")
scala> println(q.queryExecution.analyzed.stats)
Statistics(sizeInBytes=1.0 B, hints=None)

scala> println(q.queryExecution.stringifyWithStats)
== Optimized Logical Plan ==
ShowTablesCommand false, Statistics(sizeInBytes=1.0 B, hints=None)

== Physical Plan ==
Execute ShowTablesCommand
+- ShowTablesCommand false
```

```
val names = Seq((1, "one"), (2, "two")).toDF("id", "name")

// CBO is turned off by default
```

```

scala> println(spark.sessionState.conf.cboEnabled)
false

// CBO is disabled and so only sizeInBytes stat is available
// FIXME Why is analyzed required (not just logical)?
val namesStatsCboOff = names.queryExecution.analyzed.stats
scala> println(namesStatsCboOff)
Statistics(sizeInBytes=48.0 B, hints=none)

// Turn CBO on
import org.apache.spark.sql.internal.SQLConf
spark.sessionState.conf.setConf(SQLConf.CBO_ENABLED, true)

// Make sure that CBO is really enabled
scala> println(spark.sessionState.conf.cboEnabled)
true

// Invalidate the stats cache
names.queryExecution.analyzed.invalidateStatsCache

// Check out the statistics
val namesStatsCboOn = names.queryExecution.analyzed.stats
scala> println(namesStatsCboOn)
Statistics(sizeInBytes=48.0 B, hints=none)

// Despite CBO enabled, we can only get sizeInBytes stat
// That's because names is a LocalRelation under the covers
scala> println(names.queryExecution.optimizedPlan.numberedTreeString)
00 LocalRelation [id#5, name#6]

// LocalRelation triggers BasicStatsPlanVisitor to execute default case
// which is exactly as if we had CBO turned off

// Let's register names as a managed table
// That will change the rules of how stats are computed
import org.apache.spark.sql.SaveMode
names.write.mode(SaveMode.Overwrite).saveAsTable("names")

scala> spark.catalog.tableExists("names")
res5: Boolean = true

scala> spark.catalog.listTables.filter($"name" === "names").show
+-----+-----+-----+-----+
| name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|names| default|      null|   MANAGED|    false|
+-----+-----+-----+-----+

val namesTable = spark.table("names")

// names is a managed table now
// And Relation (not LocalRelation)
scala> println(namesTable.queryExecution.optimizedPlan.numberedTreeString)

```

```

00 Relation[id#32,name#33] parquet

// Check out the statistics
val namesStatsCboOn = namesTable.queryExecution.analyzed.stats
scala> println(namesStatsCboOn)
Statistics(sizeInBytes=1064.0 B, hints=None)

// Nothing has really changed, hasn't it?
// Well, sizeInBytes is bigger, but that's the only stat available
// row count stat requires ANALYZE TABLE with no NOSCAN option
sql("ANALYZE TABLE names COMPUTE STATISTICS")

// Invalidate the stats cache
namesTable.queryExecution.analyzed.invalidateStatsCache

// No change?! How so?
val namesStatsCboOn = namesTable.queryExecution.analyzed.stats
scala> println(namesStatsCboOn)
Statistics(sizeInBytes=1064.0 B, hints=None)

// Use optimized logical plan instead
val namesTableStats = spark.table("names").queryExecution.optimizedPlan.stats
scala> println(namesTableStats)
Statistics(sizeInBytes=64.0 B, rowCount=2, hints=None)

```

Note

The [statistics](#) of a Dataset are unaffected by [caching](#) it.

Note

`LogicalPlanStats` is a Scala trait with `self: LogicalPlan` as part of its definition. It is a very useful feature of Scala that restricts the set of classes that the trait could be used with (as well as makes the target subtype known at compile time).

Computing (and Caching) Statistics and Query Hints

— `stats` Method

`stats: Statistics`

`stats` gets the [statistics](#) from `statsCache` if already computed. Otherwise, `stats` branches off per whether [cost-based optimization is enabled](#) or not.

	<p>Cost-based optimization is enabled when <code>spark.sql.cbo.enabled</code> configuration property is turned on, i.e. <code>true</code>, and is disabled by default.</p> <hr/> <p>Note</p> <p>Use <code>SQLConf.cboEnabled</code> to access the current value of <code>spark.sql.cbo.enabled</code> property.</p> <pre>// CBO is disabled by default val sqlConf = spark.sessionState.conf scala> println(sqlConf.cboEnabled) false</pre>
--	---

With `cost-based optimization disabled` `stats` `requests` `SizeInBytesOnlyStatsPlanVisitor` to compute the statistics.

With `cost-based optimization enabled` `stats` `requests` `BasicStatsPlanVisitor` to compute the statistics.

In the end, `statsCache` caches the statistics for later use.

	<p><code>stats</code> is used when:</p> <ul style="list-style-type: none"> • <code>JoinSelection</code> execution planning strategy matches a logical plan: <ol style="list-style-type: none"> 1. <code>that is small enough for broadcast join</code> (using <code>BroadcastHashJoinExec</code> or <code>BroadcastNestedLoopJoinExec</code> physical operators) 2. <code>whose a single partition should be small enough to build a hash table</code> (using <code>ShuffledHashJoinExec</code> physical operator) 3. <code>that is much smaller (3X) than the other plan</code> (for <code>ShuffledHashJoinExec</code> physical operator) 4. ... • <code>QueryExecution</code> is requested for <code>stringWithStats</code> for <code>EXPLAIN COST</code> SQL command • <code>CacheManager</code> is requested to <code>cache a Dataset</code> or <code>recacheByCondition</code> • <code>HiveMetastoreCatalog</code> is requested for <code>convertToLogicalRelation</code> • <code>StarSchemaDetection</code> • <code>CostBasedJoinReorder</code> is <code>executed</code> (and does <code>reordering</code>)
--	--

Invalidating Statistics Cache (of All Operators in Logical Plan) — `invalidateStatsCache` Method

```
invalidateStatsCache(): Unit
```

`invalidateStatsCache` clears `statsCache` of the current logical operators followed by requesting the `child` logical operators for the same.

Statistics — Estimates of Plan Statistics and Query Hints

`Statistics` holds the statistics estimates and query hints of a logical operator:

- Total (output) size (in bytes)
- Estimated number of rows (aka *row count*)
- Column attribute statistics (aka *column (equi-height) histograms*)
- [Query hints](#)

Note	Cost statistics , plan statistics or query statistics are all synonyms and used interchangeably.
------	---

You can access statistics and query hints of a logical plan using `stats` property.

```
val q = spark.range(5).hint("broadcast").join(spark.range(1), "id")
val plan = q.queryExecution.optimizedPlan
val stats = plan.stats

scala> :type stats
org.apache.spark.sql.catalyst.plans.logical.Statistics

scala> println(stats.simpleString)
sizeInBytes=213.0 B, hints=none
```

Note	Use ANALYZE TABLE COMPUTE STATISTICS SQL command to compute total size and row count statistics of a table.
------	---

Note	Use ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS SQL Command to generate column (equi-height) histograms of a table.
------	---

Note	Use <code>Dataset.hint</code> or <code>SELECT</code> SQL statement with hints to specify query hints .
------	--

`Statistics` is [created](#) when:

- [Leaf logical operators](#) (specifically) and [logical operators](#) (in general) are requested for statistics estimates
- [HiveTableRelation](#) and [LogicalRelation](#) are requested for statistics estimates (through [CatalogStatistics](#))

Note	row count estimate is used in CostBasedJoinReorder logical optimization when cost-based optimization is enabled.
Note	CatalogStatistics is a "subset" of all possible <code>Statistics</code> (as there are no concepts of attributes and query hints in metastore). <code>catalogStatistics</code> are statistics stored in an external catalog (usually a Hive metastore) and are often referred as Hive statistics while <code>Statistics</code> represents the Spark statistics .

`Statistics` comes with `simpleString` method that is used for the readable **text representation** (that is `toString` with **Statistics** prefix).

```
import org.apache.spark.sql.catalyst.plans.logical.Statistics
import org.apache.spark.sql.catalyst.plans.logical.HintInfo
val stats = Statistics(sizeInBytes = 10, rowCount = Some(20), hints = HintInfo(broadcast = true))

scala> println(stats)
Statistics(sizeInBytes=10.0 B, rowCount=20, hints=(broadcast))

scala> println(stats.simpleString)
sizeInBytes=10.0 B, rowCount=20, hints=(broadcast)
```

HintInfo

`HintInfo` takes a single `broadcast` flag when created.

`HintInfo` is created when:

1. `Dataset.broadcast` function is used
2. `ResolveBroadcastHints` logical resolution rule is executed (and resolves `UnresolvedHint` logical operators)
3. `ResolvedHint` and `Statistics` are created
4. `InMemoryRelation` is requested for `computeStats` (when `sizeInBytesStats` is `0`)
5. `HintInfo` is requested to `resetForJoin`

`broadcast` is used to...FIXME

`broadcast` is off (i.e. `false`) by default.

```
import org.apache.spark.sql.catalyst.plans.logical.HintInfo
val broadcastOff = HintInfo()

scala> println(broadcastOff.broadcast)
false

val broadcastOn = broadcastOff.copy(broadcast = true)
scala> println(broadcastOn)
(broadcast)

val broadcastOff = broadcastOn.resetForJoin
scala> println(broadcastOff.broadcast)
false
```

resetForJoin Method

```
resetForJoin(): HintInfo
```

`resetForJoin` ...FIXME

Note	<code>resetForJoin</code> is used when <code>SizeInBytesOnlyStatsPlanVisitor</code> is requested to <code>visitIntersect</code> and <code>visitJoin</code> .
------	--

LogicalPlanVisitor — Contract for Computing Statistic Estimates and Query Hints of Logical Plan

`LogicalPlanVisitor` is the [contract](#) that uses the [visitor design pattern](#) to scan a logical query plan and compute [estimates of plan statistics and query hints](#).

Tip	Read about the visitor design pattern in Wikipedia .
-----	---

`LogicalPlanVisitor` defines `visit` method that dispatches computing the statistics of a logical plan to the [corresponding handler methods](#).

<pre>visit(p: LogicalPlan): T</pre>

Note	<code>T</code> stands for the type of a result to be computed (while visiting the query plan tree) and is currently always Statistics only.
------	---

The concrete `LogicalPlanVisitor` is chosen per `spark.sql.cbo.enabled` configuration property. When turned on (i.e. `true`), `LogicalPlanStats` uses [BasicStatsPlanVisitor](#) while [SizeInBytesOnlyStatsPlanVisitor](#) otherwise.

Note	<code>spark.sql.cbo.enabled</code> configuration property is off, i.e. <code>false</code> by default.
------	---

Table 1. LogicalPlanVisitors

LogicalPlanVisitor	Description
BasicStatsPlanVisitor	
SizeInBytesOnlyStatsPlanVisitor	

Table 2. LogicalPlanVisitor's Logical Operators and Their Handlers

Logical Operator	Handler
Aggregate	<code>visitAggregate</code>
Distinct	<code>visitDistinct</code>
Except	<code>visitExcept</code>
Expand	<code>visitExpand</code>
Filter	<code>visitFilter</code>
Generate	<code>visitGenerate</code>
GlobalLimit	<code>visitGlobalLimit</code>
Intersect	<code>visitIntersect</code>
Join	<code>visitJoin</code>
LocalLimit	<code>visitLocalLimit</code>
Pivot	<code>visitPivot</code>
Project	<code>visitProject</code>
Repartition	<code>visitRepartition</code>
RepartitionByExpression	<code>visitRepartitionByExpr</code>
ResolvedHint	<code>visitHint</code>
Sample	<code>visitSample</code>
ScriptTransformation	<code>visitScriptTransform</code>
Union	<code>visitUnion</code>
Window	<code>visitWindow</code>
Other logical operators	default

SizeInBytesOnlyStatsPlanVisitor — LogicalPlanVisitor for Total Size (in Bytes) Statistic Only

`SizeInBytesOnlyStatsPlanVisitor` is a [LogicalPlanVisitor](#) that computes a single dimension for [plan statistics](#), i.e. the total size (in bytes).

default Method

```
default(p: LogicalPlan): Statistics
```

Note	<code>default</code> is part of LogicalPlanVisitor Contract to compute the size statistic (in bytes) of a logical operator.
------	---

`default` requests a [leaf logical operator](#) for the statistics or creates a [Statistics](#) with the product of the `sizeInBytes` statistic of every [child operator](#).

Note	<code>default</code> uses the cache of the estimated statistics of a logical operator so the statistics of an operator is computed once until it is invalidated .
------	---

visitIntersect Method

```
visitIntersect(p: Intersect): Statistics
```

Note	<code>visitIntersect</code> is part of LogicalPlanVisitor Contract to...FIXME.
------	--

`visitIntersect` ...FIXME

visitJoin Method

```
visitJoin(p: Join): Statistics
```

Note	<code>visitJoin</code> is part of LogicalPlanVisitor Contract to...FIXME.
------	---

`visitJoin` ...FIXME

BasicStatsPlanVisitor — Computing Statistics for Cost-Based Optimization

`BasicStatsPlanVisitor` is a [LogicalPlanVisitor](#) that computes the [statistics](#) of a logical query plan for [cost-based optimization](#) (i.e. when [cost-based optimization is enabled](#)).

Note	Cost-based optimization is enabled when <code>spark.sql.cbo.enabled</code> configuration property is on, i.e. <code>true</code> , and is disabled by default.
------	---

`BasicStatsPlanVisitor` is used exclusively when a [logical operator](#) is requested for the [statistics](#) with [cost-based optimization enabled](#).

`BasicStatsPlanVisitor` comes with custom [handlers](#) for a few logical operators and falls back to [SizeInBytesOnlyStatsPlanVisitor](#) for the others.

Table 1. BasicStatsPlanVisitor's Visitor Handlers

Logical Operator	Handler	Behaviour
Aggregate	<code>visitAggregate</code>	Requests <code>AggregateEstimation</code> for statistics estimates and query hints or falls back to SizeInBytesOnlyStatsPlanVisitor
Filter	<code>visitFilter</code>	Requests <code>FilterEstimation</code> for statistics estimates and query hints or falls back to SizeInBytesOnlyStatsPlanVisitor
Join	<code>visitJoin</code>	Requests <code>JoinEstimation</code> for statistics estimates and query hints or falls back to SizeInBytesOnlyStatsPlanVisitor
Project	<code>visitProject</code>	Requests <code>ProjectEstimation</code> for statistics estimates and query hints or falls back to SizeInBytesOnlyStatsPlanVisitor

AggregateEstimation

AggregateEstimation is...FIXME

Estimating Statistics and Query Hints of Aggregate Logical Operator— estimate Method

```
estimate(agg: Aggregate): Option[Statistics]
```

estimate ...FIXME

Note

estimate is used exclusively when BasicStatsPlanVisitor is requested to estimate statistics and query hints of a Aggregate logical operator.

FilterEstimation

`FilterEstimation` is...FIXME

computeEqualityPossibilityByHistogram Internal Method

```
computeEqualityPossibilityByHistogram(literal: Literal, colStat: ColumnStat): Double
```

`computeEqualityPossibilityByHistogram` ...FIXME

Note

`computeEqualityPossibilityByHistogram` is used when...FIXME

computeComparisonPossibilityByHistogram Internal Method

```
computeComparisonPossibilityByHistogram(op: BinaryComparison, literal: Literal, colStat: ColumnStat): Double
```

`computeComparisonPossibilityByHistogram` ...FIXME

Note

`computeComparisonPossibilityByHistogram` is used when...FIXME

update Method

```
update(a: Attribute, stats: ColumnStat): Unit
```

`update` ...FIXME

Note

`update` is used when...FIXME

JoinEstimation

`JoinEstimation` is a utility that computes statistics estimates and query hints of a Join logical operator.

`JoinEstimation` is created exclusively for `BasicStatsPlanVisitor` to estimate statistics of a Join logical operator.

Note	<code>BasicStatsPlanVisitor</code> is used only when cost-based optimization is enabled.
------	--

`JoinEstimation` takes a Join logical operator when created.

When created, `JoinEstimation` immediately takes the estimated statistics and query hints of the left and right sides of the Join logical operator.

```
// JoinEstimation requires row count stats for join statistics estimates
// With cost-based optimization off, size in bytes is available only
// That would give no join estimates whatsoever (except size in bytes)
// Make sure that you `--conf spark.sql.cbo.enabled=true`
scala> println(spark.sessionState.conf.cboEnabled)
true

// Build a query with join operator
// From the available data sources tables seem the best...so far
val r1 = spark.range(5)
scala> println(r1.queryExecution.analyzed.stats.simpleString)
sizeInBytes=40.0 B, hints=none

// Make the demo reproducible
val db = spark.catalog.currentDatabase
spark.sharedState.externalCatalog.dropTable(db, table = "t1", ignoreIfNotExists = true
, purge = true)
spark.sharedState.externalCatalog.dropTable(db, table = "t2", ignoreIfNotExists = true
, purge = true)

// FIXME What relations give row count stats?

// Register tables
spark.range(5).write.saveAsTable("t1")
spark.range(10).write.saveAsTable("t2")

// Refresh internal registries
sql("REFRESH TABLE t1")
sql("REFRESH TABLE t2")

// Calculate row count stats
val tables = Seq("t1", "t2")
tables.map(t => s"ANALYZE TABLE $t COMPUTE STATISTICS").foreach(sql)
```

```

val t1 = spark.table("t1")
val t2 = spark.table("t2")

// analyzed plan is just before withCachedData and optimizedPlan plans
// where CostBasedJoinReorder kicks in and optimizes a query using statistics

val t1plan = t1.queryExecution.analyzed
scala> println(t1plan.numberedTreeString)
00 SubqueryAlias t1
01 +- Relation[id#45L] parquet

// Show the stats of every node in the analyzed query plan

val p0 = t1plan.p(0)
scala> println(s"Statistics of ${p0.simpleString}: ${p0.stats.simpleString}")
Statistics of SubqueryAlias t1: sizeInBytes=80.0 B, hints=None

val p1 = t1plan.p(1)
scala> println(s"Statistics of ${p1.simpleString}: ${p1.stats.simpleString}")
Statistics of Relation[id#45L] parquet: sizeInBytes=80.0 B, rowCount=5, hints=None

val t2plan = t2.queryExecution.analyzed

// let's get rid of the SubqueryAlias operator

import org.apache.spark.sql.catalyst.analysis.EliminateSubqueryAliases
val t1NoAliasesPlan = EliminateSubqueryAliases(t1plan)
val t2NoAliasesPlan = EliminateSubqueryAliases(t2plan)

// Using Catalyst DSL
import org.apache.spark.sql.catalyst.dsl.plans._
import org.apache.spark.sql.catalyst.plans._
val plan = t1NoAliasesPlan.join(
  otherPlan = t2NoAliasesPlan,
  joinType = Inner,
  condition = Some($"id".expr))
scala> println(plan.numberedTreeString)
00 'Join Inner, 'id
01 :- Relation[id#45L] parquet
02 +- Relation[id#57L] parquet

// Take Join operator off the logical plan
// JoinEstimation works with Joins only
import org.apache.spark.sql.catalyst.plans.logical.Join
val join = plan.collect { case j: Join => j }.head

// Make sure that row count stats are defined per join side
scala> join.left.stats.rowCount.isDefined
res1: Boolean = true

scala> join.right.stats.rowCount.isDefined
res2: Boolean = true

```

```
// Make the example reproducible
// Computing stats is once-only process and the estimates are cached
join.invalidateStatsCache

import org.apache.spark.sql.catalyst.plans.logical.statsEstimation.JoinEstimation
val stats = JoinEstimation(join).estimate
scala> :type stats
Option[org.apache.spark.sql.catalyst.plans.logical.Statistics]

// Stats have to be available so Option.get should just work
scala> println(stats.get.simpleString)
Some(sizeInBytes=1200.0 B, rowCount=50, hints=None)
```

`JoinEstimation` can estimate statistics and query hints of a Join logical operator with the following [join types](#):

- `Inner` , `Cross` , `LeftOuter` , `RightOuter` , `FullOuter` , `LeftSemi` and `LeftAnti`

For the other join types (e.g. `ExistenceJoin`), `JoinEstimation` prints out a DEBUG message to the logs and returns `None` (to "announce" that no statistics could be computed).

```
// Demo: Unsupported join type, i.e. ExistenceJoin

// Some parts were copied from the earlier demo
// FIXME Make it self-contained

// Using Catalyst DSL
// Don't even know if such existance join could ever be possible in Spark SQL
// For demo purposes it's OK, isn't it?
import org.apache.spark.sql.catalyst.plans.ExistenceJoin
val left = t1NoAliasesPlan
val right = t2NoAliasesPlan
val plan = left.join(right,
    joinType = ExistenceJoin(exists = 'id.long))

// Take Join operator off the logical plan
// JoinEstimation works with Joins only
import org.apache.spark.sql.catalyst.plans.logical.Join
val join = plan.collect { case j: Join => j }.head

// Enable DEBUG logging level
import org.apache.log4j.{Level, Logger}
Logger.getLogger("org.apache.spark.sql.catalyst.plans.logical.statsEstimation.JoinEstimation").setLevel(Level.DEBUG)

scala> val stats = JoinEstimation(join).estimate
18/06/13 10:29:37 DEBUG JoinEstimation: [CBO] Unsupported join type: ExistenceJoin(id#
35L)
stats: Option[org.apache.spark.sql.catalyst.plans.logical.Statistics] = None
```

```
// FIXME Describe the purpose of the demo

// Using Catalyst DSL
import org.apache.spark.sql.catalyst.dsl.plans._

val t1 = table(ref = "t1")

// HACK: Disable symbolToColumn implicit conversion
// It is imported automatically in spark-shell (and makes demos impossible)
// implicit def symbolToColumn(s: Symbol): org.apache.spark.sql.ColumnName
trait ThatWasABadIdea
implicit def symbolToColumn(ack: ThatWasABadIdea) = ack

import org.apache.spark.sql.catalyst.dsl.expressions._
val id = 'id.long

val t2 = table("t2")
import org.apache.spark.sql.catalyst.plans.LeftSemi
val plan = t1.join(t2, joinType = LeftSemi, condition = Some(id))
scala> println(plan.numberedTreeString)
00 'Join LeftSemi, id#2: bigint
01 :- 'UnresolvedRelation `t1`
02 +- 'UnresolvedRelation `t2`

import org.apache.spark.sql.catalyst.plans.logical.Join
val join = plan match { case j: Join => j }

import org.apache.spark.sql.catalyst.plans.logical.statsEstimation.JoinEstimation

// FIXME java.lang.UnsupportedOperationException
val stats = JoinEstimation(join).estimate
```

Tip

Enable `DEBUG` logging level for
`org.apache.spark.sql.catalyst.plans.logical.statsEstimation.JoinEstimation` logger
happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.catalyst.plans.logical.statsEstimation.JoinEst
```

Refer to [Logging](#).

estimateInnerOuterJoin Internal Method

```
estimateInnerOuterJoin(): Option[Statistics]
```

`estimateInnerOuterJoin` destructures [Join logical operator](#) into a join type with the left and right keys.

`estimateInnerOuterJoin` simply returns `None` (i.e. *nothing*) when either side of the [Join logical operator](#) have no [row count statistic](#).

Note

`estimateInnerOuterJoin` is used exclusively when `JoinEstimation` is requested to [estimate statistics and query hints of a Join logical operator](#) for `Inner`, `Cross`, `LeftOuter`, `RightOuter` and `FullOuter` joins.

computeByNdv Internal Method

```
computeByNdv(
    leftKey: AttributeReference,
    rightKey: AttributeReference,
    newMin: Option[Any],
    newMax: Option[Any]): (BigInt, ColumnStat)
```

`computeByNdv` ...FIXME

Note

`computeByNdv` is used exclusively when `JoinEstimation` is requested for [computeCardinalityAndStats](#)

computeCardinalityAndStats Internal Method

```
computeCardinalityAndStats(
    keyPairs: Seq[(AttributeReference, AttributeReference)]): (BigInt, AttributeMap[ColumnStat])
```

`computeCardinalityAndStats` ...FIXME

Note

`computeCardinalityAndStats` is used exclusively when `JoinEstimation` is requested for [estimateInnerOuterJoin](#)

**Computing Join Cardinality Using Equi-Height Histograms
— `computeByHistogram` Internal Method**

```
computeByHistogram(
    leftKey: AttributeReference,
    rightKey: AttributeReference,
    leftHistogram: Histogram,
    rightHistogram: Histogram,
    newMin: Option[Any],
    newMax: Option[Any]): (BigInt, ColumnStat)
```

`computeByHistogram` ...FIXME

Note

`computeByHistogram` is used exclusively when `JoinEstimation` is requested for `computeCardinalityAndStats` (and the histograms of both column attributes used in a join are available).

Estimating Statistics for Left Semi and Left Anti Joins — `estimateLeftSemiAntiJoin` Internal Method

```
estimateLeftSemiAntiJoin(): Option[Statistics]
```

`estimateLeftSemiAntiJoin` estimates statistics of the `Join` logical operator only when `estimated row count statistic is available`. Otherwise, `estimateLeftSemiAntiJoin` simply returns `None` (i.e. no statistics estimated).

Note

`row count` statistic of a table is available only after `ANALYZE TABLE COMPUTE STATISTICS` SQL command.

If available, `estimateLeftSemiAntiJoin` takes the `estimated row count statistic` of the `left` side of the `Join` operator.

Note

Use `ANALYZE TABLE COMPUTE STATISTICS` SQL command on the left logical plan to compute `row count` statistics.

Note

Use `ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS` SQL command on the left logical plan to generate `column (equi-height) histograms` for more accurate estimations.

In the end, `estimateLeftSemiAntiJoin` creates a new `Statistics` with the following estimates:

1. `Total size (in bytes)` is the `output size` for the `output schema` of the join, the `row count` statistic (aka `output rows`) and `column histograms`.
2. `Row count` is exactly the `row count` of the `left` side
3. `Column histograms` is exactly the `column histograms` of the `left` side

Note

`estimateLeftSemiAntiJoin` is used exclusively when `JoinEstimation` is requested to [estimate statistics and query hints](#) for `LeftSemi` and `LeftAnti` joins.

Estimating Statistics and Query Hints of Join Logical Operator— `estimate` Method

```
estimate: Option[Statistics]
```

`estimate` estimates statistics and query hints of the `Join` logical operator per `join type`:

- For `Inner`, `Cross`, `LeftOuter`, `RightOuter` and `FullOuter` join types, `estimate estimateInnerOuterJoin`
- For `LeftSemi` and `LeftAnti` join types, `estimate estimateLeftSemiAntiJoin`

For other join types, `estimate` prints out the following DEBUG message to the logs and returns `None` (to "announce" that no statistics could be computed).

```
[CBO] Unsupported join type: [joinType]
```

Note

`estimate` is used exclusively when `BasicStatsPlanVisitor` is requested to [estimate statistics and query hints of a Join logical operator](#).

ProjectEstimation

ProjectEstimation is...FIXME

Estimating Statistics and Query Hints of Project Logical Operator— estimate Method

```
estimate(project: Project): Option[Statistics]
```

estimate ...FIXME

Note

estimate is used exclusively when BasicStatsPlanVisitor is requested to estimate statistics and query hints of a Project logical operator.

Partitioning — Specification of Physical Operator's Output Partitions

`Partitioning` is the [contract](#) to hint the Spark Physical Optimizer for the number of partitions the output of a [physical operator](#) should be split across.

```
numPartitions: Int
```

`numPartitions` is used in:

- `EnsureRequirements` physical preparation rule to [enforce partition requirements of a physical operator](#)
- `SortMergeJoinExec` for `outputPartitioning` for `FullOuter` join type
- `Partitioning.allCompatible`

Table 1. Partitioning Schemes (Partitionings) and Their Properties

Partitioning	compatibleWith	guarantees	numPartitions
BroadcastPartitioning	BroadcastPartitioning with the same BroadcastMode	Exactly the same BroadcastPartitioning	1
HashPartitioning <ul style="list-style-type: none">• clustering expressions• numPartitions	HashPartitioning (when their underlying expressions are semantically equal, i.e. deterministic and canonically equal)	HashPartitioning (when their underlying expressions are semantically equal, i.e. deterministic and canonically equal)	Input numPart
PartitioningCollection <ul style="list-style-type: none">• partitionings	Any Partitioning that is compatible with one of the input partitionings	Any Partitioning that is guaranteed by any of the input partitionings	Number of partitions of the first Partitioning in the input collection
RangePartitioning <ul style="list-style-type: none">• ordering collection of SortOrder• numPartitions	RangePartitioning (when semantically equal, i.e. underlying expressions are deterministic and canonically equal)	RangePartitioning (when semantically equal, i.e. underlying expressions are deterministic and canonically equal)	Input numPart
RoundRobinPartitioning <ul style="list-style-type: none">• numPartitions	Always negative	Always negative	Input numPart
SinglePartition	Any Partitioning with exactly one partition	Any Partitioning with exactly one partition	1
UnknownPartitioning <ul style="list-style-type: none">• numPartitions	Always negative	Always negative	Input numPart

Distribution Contract — Data Distribution Across Partitions

`Distribution` is the [contract](#) of...FIXME

```
package org.apache.spark.sql.catalyst.plans.physical

sealed trait Distribution {
  def requiredNumPartitions: Option[Int]
  def createPartitioning(numPartitions: Int): Partitioning
}
```

Note	<code>Distribution</code> is a Scala <code>sealed</code> contract which means that all possible distributions are all in the same compilation unit (file).
------	--

Table 1. Distribution Contract

Method	Description		
<code>requiredNumPartitions</code>	<p>Gives the required number of partitions for a distribution.</p> <p>Used exclusively when <code>EnsureRequirements</code> physical optimization is requested to enforce partition requirements of a physical operator (and a child operator's output partitioning does not satisfy a required child distribution that leads to inserting a <code>ShuffleExchangeExec</code> operator to a physical plan).</p> <table border="1"> <tr> <td>Note</td><td> <code>None</code> for the required number of partitions indicates to use any number of partitions (possibly <code>spark.sql.shuffle.partitions</code> configuration property with the default of <code>200</code> partitions). </td></tr> </table>	Note	<code>None</code> for the required number of partitions indicates to use any number of partitions (possibly <code>spark.sql.shuffle.partitions</code> configuration property with the default of <code>200</code> partitions).
Note	<code>None</code> for the required number of partitions indicates to use any number of partitions (possibly <code>spark.sql.shuffle.partitions</code> configuration property with the default of <code>200</code> partitions).		
<code>createPartitioning</code>	<p>Creates a Partitioning for a given number of partitions.</p> <p>Used exclusively when <code>EnsureRequirements</code> physical optimization is requested to enforce partition requirements of a physical operator (and creates a <code>ShuffleExchangeExec</code> physical operator with a required <code>Partitioning</code>).</p>		

Table 2. Distributions

Distribution	Description
AllTuples	
BroadcastDistribution	
ClusteredDistribution	
HashClusteredDistribution	
OrderedDistribution	
UnspecifiedDistribution	

AllTuples

`AllTuples` is a [Distribution](#) that indicates to use one partition only.

createPartitioning Method

```
createPartitioning(numPartitions: Int): Partitioning
```

Note

`createPartitioning` is part of [Distribution Contract](#) to create a [Partitioning](#) for a given number of partitions.

```
createPartitioning ...FIXME
```

BroadcastDistribution

`BroadcastDistribution` is a [Distribution](#) that indicates to use one partition only and...FIXME.

`BroadcastDistribution` is [created](#) when:

1. `BroadcastHashJoinExec` is requested for [required child output distributions](#) (with [HashedRelationBroadcastMode](#) of the [build join keys](#))
2. `BroadcastNestedLoopJoinExec` is requested for [required child output distributions](#) (with [IdentityBroadcastMode](#))

`BroadcastDistribution` takes a [BroadcastMode](#) when created.

Note	<code>BroadcastDistribution</code> is converted to a BroadcastExchangeExec physical operator when EnsureRequirements physical query plan optimization is executed (and enforces partition requirements for data distribution and ordering).
------	---

createPartitioning Method

```
createPartitioning(numPartitions: Int): Partitioning
```

Note	<code>createPartitioning</code> is part of Distribution Contract to create a Partitioning for a given number of partitions.
------	---

`createPartitioning` ...FIXME

ClusteredDistribution

`ClusteredDistribution` is a [Distribution](#) that creates a [HashPartitioning](#) for the clustering expressions and a requested number of partitions.

`ClusteredDistribution` requires that the [clustering expressions](#) should not be empty (i.e. `Nil`).

`ClusteredDistribution` is [created](#) when the following physical operators are requested for a required child distribution:

- `MapGroupsExec`, `HashAggregateExec`, `ObjectHashAggregateExec`, `SortAggregateExec`, `WindowExec`
- Spark Structured Streaming's `FlatMapGroupsWithStateExec`, `StateStoreRestoreExec`, `StateStoreSaveExec`, `StreamingDeduplicateExec`, `StreamingSymmetricHashJoinExec`, `StreamingSymmetricHashJoinExec`
- SparkR's `FlatMapGroupsInRExec`
- PySpark's `FlatMapGroupsInPandasExec`

`ClusteredDistribution` is used when:

- `DataSourcePartitioning`, `SinglePartition`, `HashPartitioning`, and `RangePartitioning` are requested to `satisfies`
- `EnsureRequirements` is requested to [add an ExchangeCoordinator](#) for Adaptive Query Execution

createPartitioning Method

```
createPartitioning(numPartitions: Int): Partitioning
```

Note	<code>createPartitioning</code> is part of Distribution Contract to create a Partitioning for a given number of partitions.
------	---

`createPartitioning` creates a `HashPartitioning` for the [clustering expressions](#) and the input `numPartitions`.

`createPartitioning` reports an `AssertionError` when the [number of partitions](#) is not the input `numPartitions`.

This ClusteredDistribution requires [requiredNumPartitions] partitions, but the actual number of partitions is [numPartitions].

Creating ClusteredDistribution Instance

`ClusteredDistribution` takes the following when created:

- Clustering [expressions](#)
- Required number of partitions (default: `None`)

Note	<code>None</code> for the required number of partitions indicates to use any number of partitions (possibly spark.sql.shuffle.partitions configuration property with the default of <code>200</code> partitions).
------	---

HashClusteredDistribution

`HashClusteredDistribution` is a [Distribution](#) that [creates a HashPartitioning](#) for the [hash expressions](#) and a requested number of partitions.

`HashClusteredDistribution` specifies `None` for the [required number of partitions](#).

Note	<code>None</code> for the required number of partitions indicates to use any number of partitions (possibly <code>spark.sql.shuffle.partitions</code> configuration property with the default of <code>200</code> partitions).
------	--

`HashClusteredDistribution` is [created](#) when the following physical operators are requested for a required child distribution:

- `CoGroupExec`, `ShuffledHashJoinExec`, `SortMergeJoinExec`

`HashClusteredDistribution` takes hash [expressions](#) when created.

`HashClusteredDistribution` requires that the [hash expressions](#) should not be empty (i.e. `Nil`).

`HashClusteredDistribution` is used when:

- `EnsureRequirements` is requested to [add an ExchangeCoordinator](#) for Adaptive Query Execution
- `HashPartitioning` is requested to `satisfies`

createPartitioning Method

```
createPartitioning(numPartitions: Int): Partitioning
```

Note	<code>createPartitioning</code> is part of Distribution Contract to create a Partitioning for a given number of partitions.
------	---

`createPartitioning` creates a `HashPartitioning` for the [hash expressions](#) and the input `numPartitions` .

OrderedDistribution

`OrderedDistribution` is a [Distribution](#) that...FIXME

`OrderedDistribution` specifies `None` for the [required number of partitions](#).

Note	<code>None</code> for the required number of partitions indicates to use any number of partitions (possibly <code>spark.sql.shuffle.partitions</code> configuration property with the default of <code>200</code> partitions).
------	--

`OrderedDistribution` is [created](#) when...FIXME

`OrderedDistribution` takes `SortOrder` expressions for ordering when created.

`OrderedDistribution` requires that the [ordering expressions](#) should not be empty (i.e. `Nil`).

createPartitioning Method

```
createPartitioning(numPartitions: Int): Partitioning
```

Note	<code>createPartitioning</code> is part of Distribution Contract to create a Partitioning for a given number of partitions.
------	---

`createPartitioning` ...FIXME

UnspecifiedDistribution

`UnspecifiedDistribution` is a [Distribution](#) that...FIXME

`UnspecifiedDistribution` specifies `None` for the [required number of partitions](#).

Note	<code>None</code> for the required number of partitions indicates to use any number of partitions (possibly spark.sql.shuffle.partitions configuration property with the default of <code>200</code> partitions).
------	---

createPartitioning Method

```
createPartitioning(numPartitions: Int): Partitioning
```

Note	<code>createPartitioning</code> is part of Distribution Contract to create a Partitioning for a given number of partitions.
------	---

`createPartitioning` ...FIXME

Catalyst Expression — Executable Node in Catalyst Tree

`Expression` is a executable [node](#) (in a Catalyst tree) that can [evaluate](#) a result value given input values, i.e. can produce a JVM object per `InternalRow`.

Note

`Expression` is often called a **Catalyst expression** even though it is *merely* built using (not be part of) the [Catalyst — Tree Manipulation Framework](#).

```
// evaluating an expression
// Use Literal expression to create an expression from a Scala object
import org.apache.spark.sql.catalyst.expressions.Expression
import org.apache.spark.sql.catalyst.expressions.Literal
val e: Expression = Literal("hello")

import org.apache.spark.sql.catalyst.expressions.EmptyRow
val v: Any = e.eval(EmptyRow)

// Convert to Scala's String
import org.apache.spark.unsafe.types.UTF8String
scala> val s = v.asInstanceOf[UTF8String].toString
s: String = hello
```

`Expression` can [generate a Java source code](#) that is then used in evaluation.

`Expression` is **deterministic** when evaluates to the same result for the same input(s). An expression is deterministic if all the [child expressions](#) are (which for [leaf expressions](#) with no child expressions is always true).

Note

A deterministic expression is like a [pure function](#) in functional programming languages.

```
val e = $"a".expr
scala> :type e
org.apache.spark.sql.catalyst.expressions.Expression

scala> println(e.deterministic)
true
```

Note

Non-deterministic expressions are not allowed in some logical operators and are excluded in some optimizations.

`verboseString` is...FIXME

Name	Scala Kind	
BinaryExpression	abstract class	
CodegenFallback	trait	Does not support code generation and falls back to unresolved aliases .
ExpectsInputTypes	trait	
ExtractValue	trait	Marks <code>UnresolvedAliases</code> to be resolved to Aliases .
LeafExpression	abstract class	Has no child expressions (and hence "terminates")
NamedExpression		Can later be referenced in a dataflow graph.
Nondeterministic	trait	
NonSQLExpression	trait	Expression with no SQL representation Gives the only custom <code>sql</code> method that is non-overridden. When requested SQL representation , <code>NonSQLExpression</code> returns <code>None</code> .
Predicate	trait	Result data type is always boolean
TernaryExpression	abstract class	
TimeZoneAwareExpression	trait	Timezone-aware expressions
UnaryExpression	abstract class	

Unevaluable	trait	<p>Cannot be evaluated to produce a value (neither is <code>UnsupportedOperationException</code>).</p> <p>Unevaluable expressions have to be resolved (re-analyzed) by the Catalyst optimizer.</p> <pre>/** Example: Analysis failure due to an Unevaluable UnresolvedFunction is an Unevaluable expression Using Catalyst DSL to create a UnresolvedFunction */ import org.apache.spark.sql.catalyst.dsl.expressions val f = 'f.function() import org.apache.spark.sql.catalyst.dsl.plans val logicalPlan = table("t1").select(f) scala> println(logicalPlan.numberedTreeString) 00 'Project [unresolvedalias('f(), None)] 01 +- 'UnresolvedRelation `t1` scala> spark.sessionState.analyzer.execute(logicalPlan) org.apache.spark.AnalysisException: Undefined function: 'f'. at org.apache.spark.sql.catalyst.analysis.Analyzer.analyzeFunctionCall at org.apache.spark.sql.catalyst.analysis.Analyzer.analyze at org.apache.spark.sql.catalyst.analysis.package\$.analyze at org.apache.spark.sql.catalyst.analysis.Analyzer.analyze at org.apache.spark.sql.catalyst.analysis.Analyzer.analyze</pre>

Expression Contract

```
package org.apache.spark.sql.catalyst.expressions

abstract class Expression extends TreeNode[Expression] {
    // only required methods that have no implementation
    def dataType: DataType
    def doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
    def eval(input: InternalRow = EmptyRow): Any
    def nullable: Boolean
}
```

Table 2. (Subset of) Expression Contract

Method	Description
canonicalized	
checkInputDataTypes	Verifies (checks the correctness of) the input data types
childrenResolved	
dataType	Data type of the result of evaluating an expression

<code>doGenCode</code>	Code-generated expression evaluation that generates a Java source code (that is used to evaluate the expression in a more optimized way not directly using <code>eval</code>). Used when <code>Expression</code> is requested to <code>genCode</code> .		
<code>eval</code>	Interpreted (non-code-generated) expression evaluation that evaluates an expression to a JVM object for a given <code>internal binary row</code> (without generating a corresponding Java code.) <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 2px;">Note</td> <td style="padding: 2px;">By default accepts <code>EmptyRow</code>, i.e. <code>null</code>.</td> </tr> </table> <code>eval</code> is a slower "relative" of the <code>code-generated (non-interpreted) expression evaluation</code> .	Note	By default accepts <code>EmptyRow</code> , i.e. <code>null</code> .
Note	By default accepts <code>EmptyRow</code> , i.e. <code>null</code> .		
<code>foldable</code>			
<code>genCode</code>	Generates the Java source code for code-generated (non-interpreted) expression evaluation (on an input <code>internal row</code> in a more optimized way not directly using <code>eval</code>). Similar to <code>doGenCode</code> but supports expression reuse (aka <code>subexpression elimination</code>). <code>genCode</code> is a faster "relative" of the <code>interpreted (non-code-generated) expression evaluation</code> .		
<code>nullable</code>			
<code>prettyName</code>	User-facing name		
<code>references</code>			
<code>resolved</code>			
<code>semanticEquals</code>			
<code>semanticHash</code>			

reduceCodeSize Internal Method

```
reduceCodeSize(ctx: CodegenContext, eval: ExprCode): Unit
```

`reduceCodeSize` does its work only when all of the following are met:

1. Length of the generate code is above 1024
2. `INPUT_ROW` of the input `CodegenContext` is defined
3. `currentVars` of the input `CodegenContext` is not defined

Caution	FIXME When would the above not be met? What's so special about such an expression?
---------	--

`reduceCodeSize` sets the `value` of the input `ExprCode` to the [fresh term name](#) for the `value` `name`.

In the end, `reduceCodeSize` sets the code of the input `ExprCode` to the following:

```
[javaType] [newValue] = [funcFullName]([INPUT_ROW]);
```

The `funcFullName` is the [fresh term name](#) for the [name of the current expression node](#).

Tip	Use the expression node name to search for the function that corresponds to the expression in a generated code.
-----	---

Note	<code>reduceCodeSize</code> is used exclusively when <code>Expression</code> is requested to generate the Java source code for code-generated expression evaluation .
------	---

flatArguments Method

```
flatArguments: Iterator[Any]
```

`flatArguments` ...FIXME

Note	<code>flatArguments</code> is used when...FIXME
------	---

SQL Representation — sql Method

```
sql: String
```

`sql` gives a SQL representation.

Internally, `sql` gives a text representation with `prettyName` followed by `sql` of `children` in the round brackets and concatenated using the comma (,).

```
import org.apache.spark.sql.catalyst.dsl.expressions._  
import org.apache.spark.sql.catalyst.expressions.Sentences  
val sentences = Sentences("Hi there! Good morning.", "en", "US")  
  
import org.apache.spark.sql.catalyst.expressions.Expression  
val expr: Expression = count("*") === 5 && count(sentences) === 5  
scala> expr.sql  
res0: String = ((count('*') = 5) AND (count(sentences('Hi there! Good morning.', 'en', 'US')) = 5))
```

Note

sql is used when...FIXME

AggregateExpression — Unevaluable Expression Container for AggregateFunction

`AggregateExpression` is an [unevaluable expression](#) (i.e. with no support for `eval` and `doGenCode` methods) that acts as a container for an [AggregateFunction](#).

`AggregateExpression` contains the following:

- [AggregateFunction](#)
- `AggregateMode`
- `isDistinct` flag indicating whether this aggregation is distinct or not (e.g. whether SQL's `DISTINCT` keyword was used for the [aggregate function](#))
- `ExprId`

`AggregateExpression` is created when:

- Analyzer [resolves AggregateFunctions](#) (and creates an `AggregateExpression` with `Complete` aggregate mode for the functions)
- `UserDefinedAggregateFunction` is created with `isDistinct` flag [disabled](#) or [enabled](#)
- `AggUtils` is requested to [planAggregateWithOneDistinct](#) (and creates `AggregateExpressions` with `Partial` and `Final` aggregate modes for the functions)
- `Aggregator` is requested for a [TypedColumn](#) (using `Aggregator.toColumn`)
- `AggregateFunction` is [wrapped in a AggregateExpression](#)

Table 1. `toString`'s Prefixes per `AggregateMode`

Prefix	AggregateMode
<code>partial_</code>	<code>Partial</code>
<code>merge_</code>	<code>PartialMerge</code>
(empty)	<code>Final</code> OR <code>Complete</code>

Table 2. AggregateExpression's Properties

Name	Description
canonicalized	AggregateExpression with AggregateFunction expression canonicalized with the special ExprId as 0 .
children	AggregateFunction expression (for which AggregateExpression was created).
dataType	DataType of AggregateFunction expression
foldable	Disabled (i.e. false)
nullable	Whether or not AggregateFunction expression is nullable.
references	AttributeSet with the following: <ul style="list-style-type: none"> references of AggregateFunction when AggregateMode is Partial or Complete aggBufferAttributes of AggregateFunction when PartialMerge OR Final
resultAttribute	Attribute that is: <ul style="list-style-type: none"> AttributeReference when AggregateFunction is itself resolved UnresolvedAttribute otherwise
sql	Requests AggregateFunction to generate SQL output (with isDistinct flag).
toString	Prefix per AggregateMode followed by AggregateFunction's toAggString (with isDistinct flag).

AggregateFunction Contract — Aggregate Function Expressions

`AggregateFunction` is the [contract](#) for Catalyst expressions that represent **aggregate functions**.

`AggregateFunction` is used wrapped inside a [AggregateExpression](#) (using `toAggregateExpression` method) when:

- `Analyzer` [resolves functions](#) (for [SQL mode](#))
- ...FIXME: Anywhere else?

```
import org.apache.spark.sql.functions.collect_list
scala> val fn = collect_list("gid")
fn: org.apache.spark.sql.Column = collect_list(gid)

import org.apache.spark.sql.catalyst.expressions.aggregate.AggregateExpression
scala> val aggFn = fn.expr.asInstanceOf[AggregateExpression].aggregateFunction
aggFn: org.apache.spark.sql.catalyst.expressions.aggregate.AggregateFunction = collect
_list('gid, 0, 0)

scala> println(aggFn.numberedTreeString)
00 collect_list('gid, 0, 0)
01 +- 'gid
```

Note	Aggregate functions are not foldable , i.e. FIXME
------	---

Table 1. AggregateFunction Top-Level Catalyst Expressions

Name	Behaviour	Examples
DeclarativeAggregate		
ImperativeAggregate		
TypedAggregateExpression		

AggregateFunction Contract

```
abstract class AggregateFunction extends Expression {
  def aggBufferSchema: StructType
  def aggBufferAttributes: Seq[AttributeReference]
  def inputAggBufferAttributes: Seq[AttributeReference]
  def defaultResult: Option[Literal] = None
}
```

Table 2. AggregateFunction Contract

Method	Description
aggBufferSchema	<p>Schema of an aggregation buffer to hold partial aggregate results.</p> <p>Used mostly in ScalaUDAF and AggregationIterator</p>
aggBufferAttributes	<p>AttributeReferences of an aggregation buffer to hold partial aggregate results.</p> <p>Used in:</p> <ul style="list-style-type: none"> • <code>AggregateExpression</code> for references • <code>Expression</code>-based aggregate's <code>bufferSchema</code> in DeclarativeAggregate • ...
inputAggBufferAttributes	
defaultResult	Defaults to <code>None</code> .

Creating AggregateExpression for AggregateFunction — `toAggregateExpression` Method

```
toAggregateExpression(): AggregateExpression (1)
toAggregateExpression(isDistinct: Boolean): AggregateExpression
```

1. Calls the other `toAggregateExpression` with `isDistinct` disabled (i.e. `false`)

`toAggregateExpression` creates a [AggregateExpression](#) for the current `AggregateFunction` with `Complete` aggregate mode.

	<p><code>toAggregateExpression</code> is used in:</p> <ul style="list-style-type: none">• <code>functions</code> object's <code>withAggregateFunction</code> block to create a Column with AggregateExpression for a <code>AggregateFunction</code>• <code>FIXME</code>
Note	

AggregateWindowFunction Contract — Declarative Window Aggregate Function Expressions

`AggregateWindowFunction` is the [extension](#) of the [DeclarativeAggregate Contract](#) for [declarative aggregate function expressions](#) that are also [WindowFunction](#) expressions.

```
package org.apache.spark.sql.catalyst.expressions

abstract class AggregateWindowFunction extends DeclarativeAggregate with WindowFunction
{
  self: Product =>
  // No required properties (vals and methods) that have no implementation
}
```



`AggregateWindowFunction` uses [IntegerType](#) as the [data type](#) of the result of evaluating itself.

`AggregateWindowFunction` is [nullable](#) by default.

As a [WindowFunction](#) expression, `AggregateWindowFunction` uses a [SpecifiedWindowFrame](#) (with the `RowFrame` frame type, the `UnboundedPreceding` lower and the `CurrentRow` upper frame boundaries) as the [frame](#).

`AggregateWindowFunction` is a [DeclarativeAggregate](#) expression that does not support [merging](#) (two aggregation buffers together) and throws an `UnsupportedOperationException` whenever requested for it.

```
Window Functions do not support merging.
```

Table 1. `AggregateWindowFunctions` (Direct Implementations)

AggregateWindowFunction	Description
<code>RankLike</code>	
<code>RowNumberLike</code>	
<code>SizeBasedWindowFunction</code>	Window functions that require the size of the current window for calculation

AttributeReference

AttributeReference is...FIXME

Alias Unary Expression

Alias is a [unary expression](#) and a [named expression](#).

Alias is [created](#) when...FIXME

Creating Alias Instance

Alias takes the following when created:

- Child [expression](#)
- Name

Attribute — Base of Leaf Named Expressions

`Attribute` is the [base of leaf named expressions](#).

Note

[QueryPlan](#) uses `Attributes` to build the [schema](#) of the query (it represents).

```
package org.apache.spark.sql.catalyst.expressions

abstract class Attribute extends ... {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def withMetadata(newMetadata: Metadata): Attribute
    def withName(newName: String): Attribute
    def withNullability(newNullability: Boolean): Attribute
    def withQualifier(newQualifier: Option[String]): Attribute
    def newInstance(): Attribute
}
```

Table 1. Attribute Contract

Property	Description
<code>withMetadata</code>	
<code>withName</code>	
<code>withNullability</code>	
<code>withQualifier</code>	
<code>newInstance</code>	

When requested for [references](#), `Attribute` gives the reference to itself only.

As a [NamedExpression](#), `Attribute` gives the reference to itself only when requested for [toAttribute](#).

Table 2. Attributes (Direct Implementations)

Attribute	Description
AttributeReference	
PrettyAttribute	
UnresolvedAttribute	

As an optimization, `Attribute` is marked as to not tolerate `nulls`, and when given a `null` input produces a `null` output.

BoundReference Leaf Expression — Reference to Value in Internal Binary Row

`BoundReference` is a leaf expression that evaluates to a value in an internal binary row at a specified position and of a given data type.

`BoundReference` takes the following when created:

- Ordinal, i.e. the position
- Data type of the value
- `nullable` flag that controls whether the value can be `null` or not

```
import org.apache.spark.sql.catalyst.expressions.BoundReference
import org.apache.spark.sql.types.LongType
val boundRef = BoundReference(ordinal = 0, dataType = LongType, nullable = true)

scala> println(boundRef.toString)
input[0, bigint, true]

import org.apache.spark.sql.catalyst.InternalRow
val row = InternalRow(1L, "hello")

val value = boundRef.eval(row).asInstanceOf[Long]
```

You can also create a `BoundReference` using Catalyst DSL's `at` method.

```
import org.apache.spark.sql.catalyst.dsl.expressions._
val boundRef = 'hello.string.at(4)
scala> println(boundRef)
input[4, string, true]
```

Evaluating Expression — `eval` Method

```
eval(input: InternalRow): Any
```

Note

`eval` is part of [Expression Contract](#) for the **interpreted (non-code-generated) expression evaluation**, i.e. evaluating a Catalyst expression to a JVM object for a given [internal binary row](#).

`eval` gives the value at position from the input internal binary row that is of a correct type.

Internally, `eval` returns `null` if the value at the `position` is `null`.

Otherwise, `eval` uses the methods of `InternalRow` per the defined `data type` to access the value.

Table 1. eval's `DataType` to `InternalRow`'s Methods Mapping (in execution order)

DataType	InternalRow's Method
<code>BooleanType</code>	<code>getBoolean</code>
<code>ByteType</code>	<code>getByte</code>
<code>ShortType</code>	<code>getShort</code>
<code>IntegerType</code> or <code>DateType</code>	<code>getInt</code>
<code>LongType</code> or <code>TimestampType</code>	<code>getLong</code>
<code>FloatType</code>	<code>getFloat</code>
<code>DoubleType</code>	<code>getDouble</code>
<code>StringType</code>	<code>getUTF8String</code>
<code>BinaryType</code>	<code>getBinary</code>
<code>CalendarIntervalType</code>	<code>getInterval</code>
<code>DecimalType</code>	<code>getDecimal</code>
<code>StructType</code>	<code>getStruct</code>
<code>ArrayType</code>	<code>getArray</code>
<code>MapType</code>	<code>getMap</code>
<code>others</code>	<code>get(ordinal, dataType)</code>

Generating Java Source Code (ExprCode) For Code-Generated Expression Evaluation — doGenCode Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

`doGenCode` is part of [Expression Contract](#) to generate a Java source code (`ExprCode`) for code-generated expression evaluation.

`doGenCode` ...FIXME

BindReferences.bindReference Method

```
bindReference[A <: Expression](
  expression: A,
  input: AttributeSeq,
  allowFailures: Boolean = false): A
```

`bindReference` ...FIXME

Note

`bindReference` is used when...FIXME

CallMethodViaReflection Expression

`CallMethodViaReflection` is an [expression](#) that represents a static method call in Scala or Java using `reflect` and `java_method` functions.

Note

`reflect` and `java_method` functions are only supported in [SQL](#) and [expression](#) modes.

Table 1. CallMethodViaReflection's DataType to JVM Types Mapping

DataType	JVM Type
<code>BooleanType</code>	<code>java.lang.Boolean</code> / <code>scala.Boolean</code>
<code>ByteType</code>	<code>java.lang.Byte</code> / <code>Byte</code>
<code>ShortType</code>	<code>java.lang.Short</code> / <code>Short</code>
<code>IntegerType</code>	<code>java.lang.Integer</code> / <code>Int</code>
<code>LongType</code>	<code>java.lang.Long</code> / <code>Long</code>
<code>FloatType</code>	<code>java.lang.Float</code> / <code>Float</code>
<code>DoubleType</code>	<code>java.lang.Double</code> / <code>Double</code>
<code>StringType</code>	<code>String</code>

```

import org.apache.spark.sql.catalyst.expressions.CallMethodViaReflection
import org.apache.spark.sql.catalyst.expressions.Literal
scala> val expr = CallMethodViaReflection(
|   Literal("java.time.LocalDateTime") :::
|   Literal("now") :: Nil)
expr: org.apache.spark.sql.catalyst.expressions.CallMethodViaReflection = reflect(java
.time.LocalDateTime, now)
scala> println(expr.numberedTreeString)
00 reflect(java.time.LocalDateTime, now)
01 :- java.time.LocalDateTime
02 +- now

// CallMethodViaReflection as the expression for reflect SQL function
val q = """
  select reflect("java.time.LocalDateTime", "now") as now
"""
val plan = spark.sql(q).queryExecution.logical
// CallMethodViaReflection shows itself under "reflect" name
scala> println(plan.numberedTreeString)
00 Project [reflect(java.time.LocalDateTime, now) AS now#39]
01 +- OneRowRelation$
```

`CallMethodViaReflection` supports a [fallback mode for expression code generation](#).

Table 2. `CallMethodViaReflection`'s Properties

Property	Description
<code>dataType</code>	<code>StringType</code>
<code>deterministic</code>	Disabled (i.e. <code>false</code>)
<code>nullable</code>	Enabled (i.e. <code>true</code>)
<code>prettyName</code>	<code>reflect</code>

Note

`CallMethodViaReflection` is very similar to [StaticInvoke](#) expression.

Coalesce Expression

`Coalesce` is a [Catalyst expression](#) to represent `coalesce` standard function or SQL's `coalesce` function in structured queries.

When created, `Coalesce` takes [Catalyst expressions](#) (as the children).

```
import org.apache.spark.sql.catalyst.expressions.Coalesce

// Use Catalyst DSL
import org.apache.spark.sql.catalyst.dsl.expressions._

import org.apache.spark.sql.functions.lit
val coalesceExpr = Coalesce(children = Seq(lit(null).expr % 1, lit(null).expr, 1d))
scala> println(coalesceExpr.numberedTreeString)
00 coalesce((null % 1), null, 1.0)
01 :- (null % 1)
02 : :- null
03 : +- 1
04 :- null
05 +- 1.0
```

Caution

FIXME Describe FunctionArgumentConversion and Coalesce

Spark Optimizer uses [NullPropagation](#) logical optimization to remove `null` literals (in the `children` expressions). That could result in a static evaluation that gives `null` value if all `children` expressions are `null` literals.

```
// FIXME
// Demo Coalesce with nulls only
// Demo Coalesce with null and non-null expressions that are optimized to one expression (in NullPropagation)
// Demo Coalesce with non-null expressions after NullPropagation optimization
```

`Coalesce` is also [created](#) when:

- `Analyzer` is requested to [commonNaturalJoinProcessing](#) for `FullOuter` join type
- `RewriteDistinctAggregates` logical optimization is requested to `rewrite`
- `ExtractEquiJoinKeys` Scala extractor is requested to [destructure a logical plan](#)
- `ColumnStat` is requested to [statExprs](#)
- `IfNull` expression is created

- `Nvl` expression is created
- Whenever `cast` expression is used in Catalyst expressions (e.g. `Average` , `sum`)

CodegenFallback Contract — Catalyst Expressions with Fallback Code Generation Mode

`CodegenFallback` is the [contract](#) of Catalyst expressions that do not support a Java code generation and want to [fall back to interpreted mode](#) (aka *fallback mode*).

`CodegenFallback` is used when `CollapseCodegenStages` physical optimization is requested to [execute](#) (and [enforce whole-stage codegen requirements for Catalyst expressions](#)).

```
package org.apache.spark.sql.catalyst.expressions.codegen

trait CodegenFallback extends Expression {
    // No properties (vals and methods) that have no implementation
}
```

Table 1. (Some Examples of) CodegenFallbacks

CodegenFallback	Description
<code>CurrentDate</code>	
<code>CurrentTimestamp</code>	
<code>Cube</code>	
JsonToStructs	
<code>Rollup</code>	
<code>StructsToJson</code>	

Example — `CurrentTimestamp` Expression with nullable flag disabled

```

import org.apache.spark.sql.catalyst.expressions.CurrentTimestamp
val currTimestamp = CurrentTimestamp()

import org.apache.spark.sql.catalyst.expressions.codegen.CodegenFallback
assert(currTimestamp.isInstanceOf[CodegenFallback], "CurrentTimestamp should be a CodegenFallback")

assert(currTimestamp.nullable == false, "CurrentTimestamp should not be nullable")

import org.apache.spark.sql.catalyst.expressions.codegen.{CodegenContext, ExprCode}
val ctx = new CodegenContext
// doGenCode is used when Expression.genCode is executed
val ExprCode(code, _, _) = currTimestamp.genCode(ctx)

scala> println(code)

Object obj_0 = ((Expression) references[0]).eval(null);
    long value_0 = (Long) obj_0;

```

Generating Java Source Code (ExprCode) For Code-Generated Expression Evaluation — doGenCode Method

doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode

Note

doGenCode is part of [Expression Contract](#) to generate a Java source code (ExprCode) for code-generated expression evaluation.

doGenCode requests the input CodegenContext to add itself to the [references](#).

doGenCode walks down the expression tree to find [Nondeterministic](#) expressions and for every [Nondeterministic](#) expression does the following:

1. Requests the input CodegenContext to add it to the [references](#)
2. Requests the input CodegenContext to [addPartitionInitializationStatement](#) that is a Java code block as follows:

```
((Nondeterministic) references[[childIndex]])
    .initialize(partitionIndex);
```

In the end, doGenCode generates a plain Java source code block that is one of the following code blocks per the [nullable](#) flag. doGenCode copies the input ExprCode with the code block added (as the [code](#) property).

doGenCode Code Block for nullable flag enabled

```
[placeHolder]
Object [objectTerm] = ((Expression) references[[idx]]).eval([input]);
boolean [isNull] = [objectTerm] == null;
[javaType] [value] = [defaultValue];
if (![isNull]) {
    [value] = ([boxedType]) [objectTerm];
}
```

doGenCode Code Block for nullable flag disabled

```
[placeHolder]
Object [objectTerm] = ((Expression) references[[idx]]).eval([input]);
[javaType] [value] = ([boxedType]) [objectTerm];
```

CollectionGenerator Generator Expression Contract

`CollectionGenerator` is the [contract](#) in Spark SQL for Generator expressions that [generate](#) a collection object (i.e. an array or map) and (at execution time) [use a different path for whole-stage Java code generation](#) (while executing `GenerateExec` physical operator with Whole-Stage Java Code Generation enabled).

```
package org.apache.spark.sql.catalyst.expressions

trait CollectionGenerator extends Generator {
  def collectionType: DataType = dataType
  def inline: Boolean
  def position: Boolean
}
```

Table 1. CollectionGenerator Contract

Method	Description
<code>collectionType</code>	The type of the returned collection object. Used when...
<code>inline</code>	Flag whether to inline rows during whole-stage Java code generation. Used when...
<code>position</code>	Flag whether to include the positions of elements within the result collection. Used when...

Table 2. CollectionGenerators

CollectionGenerator	Description
Inline	
ExplodeBase	
Explode	
PosExplode	

ComplexTypedAggregateExpression

ComplexTypedAggregateExpression is...FIXME

ComplexTypedAggregateExpression is [created](#) when...FIXME

Creating ComplexTypedAggregateExpression Instance

ComplexTypedAggregateExpression takes the following when created:

- [Aggregator](#)
- Optional input deserializer [expression](#)
- Optional Java class for the input
- Optional [schema](#) for the input
- Buffer serializer (as a collection of [named expressions](#))
- Buffer deserializer [expression](#)
- Output serializer (as a collection of [expressions](#))
- [DataType](#)
- `nullable` flag
- `mutableAggBufferOffset` (default: `0`)
- `inputAggBufferOffset` (default: `0`)

CreateArray

CreateArray is...FIXME

CreateNamedStruct Expression

`CreateNamedStruct` is a [CreateNamedStructLike](#) expression that...FIXME

`CreateNamedStruct` uses **named_struct** for the [user-facing name](#).

```
// Using Catalyst DSL
import org.apache.spark.sql.catalyst.dsl.expressions._
val s = namedStruct("*)
scala> println(s)
named_struct(*)
```

`CreateNamedStruct` is registered in [FunctionRegistry](#) under the name of `named_struct` SQL function.

```
import org.apache.spark.sql.catalyst.FunctionIdentifier
val fid = FunctionIdentifier(funcName = "named_struct")
val className = spark.sessionState.functionRegistry.lookupFunction(fid).get.getClassName
scala> println(className)
org.apache.spark.sql.catalyst.expressions.CreateNamedStruct

val q = sql("SELECT named_struct('id', 0)")
// analyzed so the function is resolved already (using FunctionRegistry)
val analyzedPlan = q.queryExecution.analyzed
scala> println(analyzedPlan.numberedTreeString)
00 Project [named_struct(id, 0) AS named_struct(id, 0)#7]
01 +- OneRowRelation

val e = analyzedPlan.expressions.head.children.head
import org.apache.spark.sql.catalyst.expressions.CreateNamedStruct
assert(e.isInstanceOf[CreateNamedStruct])
```

`CreateNamedStruct` is [created](#) when:

- [ScalaReflection](#), [RowEncoder](#) and `JavaTypeInference` are requested for a serializer of a type
- [TimeWindowing](#) and [ResolveCreateNamedStruct](#) logical resolution rules are executed
- `CreateStruct` is requested to [create a CreateNamedStruct expression](#)

`CreateNamedStruct` takes a collection of [Catalyst](#) expressions when created.

`CreateNamedStruct` generates Java source code (as `ExprCode`) for code-generated expression evaluation.

```
// You could also use Seq("*")
import org.apache.spark.sql.functions.lit
val exprs = Seq("a", 1).map(lit).map(_.expr)

import org.apache.spark.sql.catalyst.expressions.CreateNamedStruct
val createNamedStruct = CreateNamedStruct(exprs)

import org.apache.spark.sql.catalyst.expressions.codegen.{CodegenContext, ExprCode}
val ctx = new CodegenContext
// doGenCode is used when Expression.genCode is executed
val ExprCode(code, _, _) = createNamedStruct.genCode(ctx)

// Helper methods
def trim(code: String): String = {
  code.trim.split("\n").map(_.trim).filter(line => line.nonEmpty).mkString("\n")
}
def prettyPrint(code: String) = println(trim(code))
// END: Helper methods

scala> println(trim(code))
Object[] values_0 = new Object[1];
if (false) {
values_0[0] = null;
} else {
values_0[0] = 1;
}
final InternalRow value_0 = new org.apache.spark.sql.catalyst.expressions.GenericInternalRow(values_0);
values_0 = null;
```

Use `namedStruct` operator from Catalyst DSL's [expressions](#) to create a `CreateNamedStruct` expression.

Tip

```
import org.apache.spark.sql.catalyst.dsl.expressions.-
val s = namedStruct()
scala> :type s
org.apache.spark.sql.catalyst.expressions.Expression
import org.apache.spark.sql.catalyst.expressions.CreateNamedStruct
assert(s.isInstanceOf[CreateNamedStruct])

val s = namedStruct("*)
scala> println(s)
named_struct(*)
```

Generating Java Source Code (ExprCode) For Code-Generated Expression Evaluation — `doGenCode` Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

`doGenCode` is part of [Expression Contract](#) to generate a Java source code (`ExprCode`) for code-generated expression evaluation.

`doGenCode ...FIXME`

CreateNamedStructLike Contract

`CreateNamedStructLike` is the base of [Catalyst expressions](#) that [FIXME](#).

```
package org.apache.spark.sql.catalyst.expressions

trait CreateNamedStructLike extends Expression {
    // no required properties (vals and methods) that have no implementation
}
```

`CreateNamedStructLike` is not [nullable](#).

`CreateNamedStructLike` is [foldable](#) only if all [value expressions](#) are.

Table 1. CreateNamedStructLikes (Direct Implementations)

CreateNamedStructLike	Description
<code>CreateNamedStruct</code>	
<code>CreateNamedStructUnsafe</code>	

Table 2. CreateNamedStructLike's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>dataType</code>	
<code>nameExprs</code>	Catalyst expressions for names
<code>names</code>	
<code>valExprs</code>	Catalyst expressions for values

Checking Input Data Types — `checkInputDataTypes` Method

```
checkInputDataTypes(): TypeCheckResult
```

Note	<code>checkInputDataTypes</code> is part of the Expression Contract to verify (check the correctness of) the input data types.
------	--

`checkInputDataTypes` ...[FIXME](#)

Evaluating Expression — eval Method

```
eval(input: InternalRow): Any
```

Note

`eval` is part of [Expression Contract](#) for the **interpreted (non-code-generated) expression evaluation**, i.e. evaluating a Catalyst expression to a JVM object for a given [internal binary row](#).

`eval` ...FIXME

CreateNamedStructUnsafe Expression

`CreateNamedStructUnsafe` is a [CreateNamedStructLike](#) expression that...FIXME

Generating Java Source Code (`ExprCode`) For Code-Generated Expression Evaluation — `doGenCode` Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

`doGenCode` is part of [Expression Contract](#) to generate a Java source code (`ExprCode`) for code-generated expression evaluation.

`doGenCode` ...FIXME

CumeDist Declarative Window Aggregate Function Expression

`CumeDist` is a [SizeBasedWindowFunction](#) and a [RowNumberLike](#) expression that is used for the following:

- [cume_dist](#) standard function (Dataset API)
- [cume_dist](#) SQL function

`CumeDist` takes no input parameters when created.

```
import org.apache.spark.sql.catalyst.expressions.CumeDist
val cume_dist = CumeDist()
```

`CumeDist` uses [cume_dist](#) for the user-facing name.

```
import org.apache.spark.sql.catalyst.expressions.CumeDist
val cume_dist = CumeDist()
scala> println(cume_dist)
cume_dist()
```

As an [WindowFunction](#) expression (indirectly), `CumeDist` requires the `SpecifiedWindowFrame` (with the `RangeFrame` frame type, the `UnboundedPreceding` lower and the `CurrentRow` upper frame boundaries) as the `frame`.

Note	The frame for <code>CumeDist</code> expression is range-based instead of row-based, because it has to return the same value for tie values in a window (equal values per <code>ORDER BY</code> specification).
------	--

As a [DeclarativeAggregate](#) expression (indirectly), `cumeDist` defines the `evaluateExpression` expression which returns the final value when `CumeDist` is evaluated. The value uses the formula `rowNumber / n` where `rowNumber` is the row number in a window frame (the number of values before and including the current row) divided by the number of rows in the window frame.

```
import org.apache.spark.sql.catalyst.expressions.CumeDist
val cume_dist = CumeDist()
scala> println(cume_dist.evaluateExpression.numberedTreeString)
00 (cast(rowNumber#0 as double) / cast(window_partition_size#1 as double))
01 :- cast(rowNumber#0 as double)
02 : +- rowNumber#0: int
03 +- cast(window_partition_size#1 as double)
04   +- window_partition_size#1: int
```

DeclarativeAggregate Contract — Unevaluable Aggregate Function Expressions

`DeclarativeAggregate` is an extension of the [AggregateFunction Contract](#) for aggregate function expressions that are [unevaluable](#) and use expressions for evaluation.

Note	An unevaluable expression cannot be evaluated to produce a value (neither in interpreted nor code-generated expression evaluations) and has to be resolved (replaced) to some other expressions or logical operators at analysis or optimization phases or they fail analysis.
------	--

Table 1. DeclarativeAggregate Contract

Property	Description
<code>evaluateExpression</code>	<pre>evaluateExpression: Expression</pre> <p>The expression that returns the final value for the aggregate function</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>AggregationIterator</code> is requested for the generateResultProjection • <code>HashAggregateExec</code> physical operator is requested to doProduceWithoutKeys and generateResultFunction • <code>AggregateProcessor</code> is created (when <code>WindowExec</code> physical operator is executed)
<code>initialValues</code>	<code>initialValues: Seq[Expression]</code>
<code>mergeExpressions</code>	<code>mergeExpressions: Seq[Expression]</code>
<code>updateExpressions</code>	<code>updateExpressions: Seq[Expression]</code>

Table 2. DeclarativeAggregates (Direct Implementations)

DeclarativeAggregate	Description
AggregateWindowFunction	Contract for declarative window aggregate function expressions
Average	
CentralMomentAgg	
Corr	
Count	
Covariance	
First	
Last	
Max	
Min	
SimpleTypedAggregateExpression	
Sum	

ExecSubqueryExpression Contract — Catalyst Expressions with SubqueryExec Physical Operators

`ExecSubqueryExpression` is the [contract](#) for Catalyst expressions that contain a physical plan with [SubqueryExec](#) physical operator (i.e. `PlanExpression[SubqueryExec]`).

```
package org.apache.spark.sql.execution

abstract class ExecSubqueryExpression extends PlanExpression[SubqueryExec] {
  def updateResult(): Unit
}
```

Table 1. ExecSubqueryExpression Contract

Method	Description
<code>updateResult</code>	Used exclusively when a physical operator is requested to waitForSubqueries (when executed as part of Physical Operator Execution Pipeline).

Table 2. ExecSubqueryExpressions

ExecSubqueryExpression	Description
InSubquery	
ScalarSubquery	

Exists — Correlated Predicate Subquery Expression

`Exists` is a [SubqueryExpression](#) and a [predicate expression](#) (i.e. the result [data type](#) is always [boolean](#)).

`Exists` is [created](#) when:

1. `ResolveSubquery` is requested to [resolveSubQueries](#)
2. `PullupCorrelatedPredicates` is requested to [rewriteSubQueries](#)
3. `AstBuilder` is requested to [visitExists](#) (in SQL statements)

`Exists` [cannot be evaluated](#), i.e. produce a value given an internal row.

When requested to evaluate or `doGenCode`, `Exists` simply reports a `UnsupportedOperationException`.

```
Cannot evaluate expression: [this]
```

`Exists` is never [nullable](#).

`Exists` uses the following **text representation**:

```
exists#[exprId] [conditionString]
```

When requested for a [canonicalized](#) version, `Exists` creates a new instance with...FIXME

Creating Exists Instance

`Exists` takes the following when created:

- Subquery [logical plan](#)
- Child [expressions](#)
- `ExprId`

ExpectsInputTypes Contract

ExpectsInputTypes ...FIXME

ExplodeBase Base Generator Expression

`ExplodeBase` is the base class for [Explode](#) and [PosExplode](#) generator expressions.

`ExplodeBase` is a [unary expression](#) and [Generator](#) with [CodegenFallback](#).

Explode Generator Unary Expression

`Explode` is a unary expression that produces a sequence of records for each value in the array or map.

`Explode` is a result of executing `explode` function (in SQL and [functions](#))

```
scala> sql("SELECT explode(array(10,20))").explain
== Physical Plan ==
Generate explode([10,20]), false, false, [col#68]
+- Scan OneRowRelation[]

scala> sql("SELECT explode(array(10,20))").queryExecution.optimizedPlan.expressions(0)
res18: org.apache.spark.sql.catalyst.expressions.Expression = explode([10,20])

val arrayDF = Seq(Array(0,1)).toDF("array")
scala> arrayDF.withColumn("num", explode('array)).explain
== Physical Plan ==
Generate explode(array#93), true, false, [array#93, num#102]
+- LocalTableScan [array#93]
```

PosExplode

Caution	FIXME
---------	-------

First Aggregate Function Expression

`First` is a [DeclarativeAggregate](#) function expression that is [created](#) when:

- `AstBuilder` is requested to [parse a FIRST statement](#)
- `first` standard function is used
- `first` and `first_value` [SQL functions](#) are used

```
val sqlText = "FIRST (organizationName IGNORE NULLS)"
val e = spark.sessionState.sqlParser.parseExpression(sqlText)
scala> :type e
org.apache.spark.sql.catalyst.expressions.Expression

import org.apache.spark.sql.catalyst.expressions.aggregate.AggregateExpression
val aggExpr = e.asInstanceOf[AggregateExpression]

import org.apache.spark.sql.catalyst.expressions.aggregate.First
val f = aggExpr.aggregateFunction
scala> println(f.simpleString)
first('organizationName) ignore nulls
```

When requested to [evaluate](#) (and return the final value), `First` simply returns a [AttributeReference](#) (with `first` name and the [data type](#) of the `child` expression).

Tip

Use `first` operator from the [Catalyst DSL](#) to create an `First` aggregate function expression, e.g. for testing or Spark SQL internals exploration.

Catalyst DSL — `first` Operator

```
first(e: Expression): Expression
```

`first` [creates](#) a `First` expression and requests it to [convert to a AggregateExpression](#).

```
import org.apache.spark.sql.catalyst.dsl.expressions._  
val e = first('orgName)  
  
scala> println(e.numberedTreeString)  
00 first('orgName, false)  
01 +- first('orgName)()  
02   :- 'orgName  
03   +- false  
  
import org.apache.spark.sql.catalyst.expressions.aggregate.AggregateExpression  
val aggExpr = e.asInstanceOf[AggregateExpression]  
  
import org.apache.spark.sql.catalyst.expressions.aggregate.First  
val f = aggExpr.aggregateFunction  
scala> println(f.simpleString)  
first('orgName)()
```

Creating First Instance

`First` takes the following when created:

- Child `expression`
- `ignoreNullsExpr` flag `expression`

Generator Contract — Expressions to Generate Zero Or More Rows (aka Lateral Views)

`Generator` is a [contract](#) for [Catalyst expressions](#) that can [produce](#) zero or more rows given a single input row.

Note	<code>Generator</code> corresponds to SQL's LATERAL VIEW .
------	--

`dataType` in `Generator` is simply an [ArrayType](#) of [elementSchema](#).

`Generator` is not [foldable](#) and not [nullable](#) by default.

`Generator` supports [Java code generation](#) (aka *whole-stage codegen*) conditionally, i.e. only when a physical operator is not marked as [CodegenFallback](#).

`Generator` uses `terminate` to inform that there are no more rows to process, clean up code, and additional rows can be made here.

```
terminate(): TraversableOnce[InternalRow] = Nil
```

Table 1. Generators

Name	Description		
CollectionGenerator			
ExplodeBase			
Explode			
GeneratorOuter			
HiveGenericUDTF			
Inline	Corresponds to <code>inline</code> and <code>inline_outer</code> functions.		
JsonTuple			
PosExplode			
Stack			
UnresolvedGenerator	<p>Represents an unresolved generator.</p> <p>Created when <code>AstBuilder</code> creates generate unary logical operator for <code>LATERAL VIEW</code> that corresponds to the following:</p> <pre>LATERAL VIEW (OUTER)? generatorFunctionName (arg1, arg2, ...) tblName AS? col1, col2, ...</pre> <table border="1" style="margin-left: 20px;"> <tr> <td style="padding: 5px;">Note</td> <td style="padding: 5px;">unresolvedGenerator is resolved to Generator by <code>ResolveFunctions</code> logical evaluation rule.</td> </tr> </table>	Note	unresolvedGenerator is resolved to Generator by <code>ResolveFunctions</code> logical evaluation rule.
Note	unresolvedGenerator is resolved to Generator by <code>ResolveFunctions</code> logical evaluation rule.		
UserDefinedGenerator	Used exclusively in the deprecated <code>explode</code> operator		

You can only have one generator per select clause that is enforced by `ExtractGen`.

```
scala> xys.select(explode($"xs"), explode($"ys")).show
org.apache.spark.sql.AnalysisException: Only one generator allowed per select c
  at org.apache.spark.sql.catalyst.analysis.Analyzer$ExtractGenerator$$anonfun$1
  at org.apache.spark.sql.catalyst.analysis.Analyzer$ExtractGenerator$$anonfun$1
  at org.apache.spark.sql.catalyst.plans.logical.LogicalPlan$$anonfun$resolveOp
```

If you want to have more than one generator in a structured query you should use e.g.

```
val arrayTuple = (Array(1,2,3), Array("a", "b", "c"))
val ncs = Seq(arrayTuple).toDF("ns", "cs")
```

```
scala> ncs.show
+-----+
|       ns |      cs |
+-----+
|[1, 2, 3]| [a, b, c]
+-----+
```

Note

```
scala> ncs.createOrReplaceTempView("ncs")
```

```
val q = """
    SELECT n, c FROM ncs
    LATERAL VIEW explode(ns) nsExpl AS n
    LATERAL VIEW explode(cs) csExpl AS c
""";
```

```
scala> sql(q).show
+---+---+
| n | c |
+---+---+
| 1 | a |
| 1 | b |
| 1 | c |
| 2 | a |
| 2 | b |
| 2 | c |
| 3 | a |
| 3 | b |
| 3 | c |
+---+---+
```

Generator Contract

```
package org.apache.spark.sql.catalyst.expressions

trait Generator extends Expression {
    // only required methods that have no implementation
    def elementSchema: StructType
    def eval(input: InternalRow): TraversableOnce[InternalRow]
}
```

Table 2. (Subset of) Generator Contract

Method	Description
elementSchema	Schema of the elements to be generated
eval	

GetArrayStructFields

GetArrayStructFields is...FIXME

GetArrayItem

GetArrayItem is...FIXME

GetMapView

GetMapView is...FIXME

GetStructField Unary Expression

`GetStructField` is a [unary expression](#) that...FIXME

`GetStructField` is [created](#) when...FIXME

Creating GetStructField Instance

`GetStructField` takes the following when created:

- Child [expression](#)
- Ordinal
- Optional name

ImperativeAggregate — Contract for Aggregate Function Expressions with Imperative Methods

`ImperativeAggregate` is the [contract](#) for [aggregate functions](#) that are expressed in terms of imperative [initialize](#), [update](#), and [merge](#) methods (that operate on `Row`-based aggregation buffers).

`ImperativeAggregate` is a [Catalyst expression](#) with [CodegenFallback](#).

Table 1. ImperativeAggregate's Direct Implementations

Name	Description
<code>HyperLogLogPlusPlus</code>	
<code>PivotFirst</code>	
<code>ScalaUDAF</code>	
<code>TypedImperativeAggregate</code>	

ImperativeAggregate Contract

```
package org.apache.spark.sql.catalyst.expressions.aggregate

abstract class ImperativeAggregate {
  def initialize(mutableAggBuffer: InternalRow): Unit
  val inputAggBufferOffset: Int
  def merge(mutableAggBuffer: InternalRow, inputAggBuffer: InternalRow): Unit
  val mutableAggBufferOffset: Int
  def update(mutableAggBuffer: InternalRow, inputRow: InternalRow): Unit
  def withNewInputAggBufferOffset(newInputAggBufferOffset: Int): ImperativeAggregate
  def withNewMutableAggBufferOffset(newMutableAggBufferOffset: Int): ImperativeAggregate
}
```

Table 2. ImperativeAggregate Contract

Method	Description
<code>initialize</code>	<p>Used when:</p> <ul style="list-style-type: none"> • <code>AggregateProcessor</code> is initialized (for window aggregate functions) • AggregationIterator, ObjectAggregationIterator, TungstenAggregationIterator (for aggregate functions)
<code>inputAggBufferOffset</code>	
<code>merge</code>	<p>Used when:</p> <ul style="list-style-type: none"> • <code>AggregationIterator</code> does generateProcessRow (for aggregate functions)
<code>mutableAggBufferOffset</code>	
<code>update</code>	<p>Used when:</p> <ul style="list-style-type: none"> • <code>AggregateProcessor</code> is updated (for window aggregate functions) • AggregationIterator (for aggregate functions)
<code>withNewInputAggBufferOffset</code>	
<code>withNewMutableAggBufferOffset</code>	

In Predicate Expression

`In` is a [predicate expression](#) (i.e. the result [data type](#) is always [boolean](#)).

`In` is [created](#) when:

- `Column.isin` operator is used
- `AstBuilder` is requested to [parse SQL's IN predicate with a subquery](#)

Tip

Use Catalyst DSL's `in` operator to create an `In` expression.

```
in(list: Expression*): Expression
```

```
// Using Catalyst DSL to create an In expression
import org.apache.spark.sql.catalyst.dsl.expressions._

// HACK: Disable symbolToColumn implicit conversion
// It is imported automatically in spark-shell (and makes demos impossible)
// implicit def symbolToColumn(s: Symbol): org.apache.spark.sql.ColumnName
trait ThatWasABadIdea
implicit def symbolToColumn(ack: ThatWasABadIdea) = ack

val value = 'a.long
import org.apache.spark.sql.catalyst.expressions.{Expression, Literal}
import org.apache.spark.sql.types.StringType
val list: Seq[Expression] = Seq(1, Literal.create(null, StringType), true)

val e = value in (list: _*)

scala> :type e
org.apache.spark.sql.catalyst.expressions.Expression

scala> println(e.dataType)
BooleanType

scala> println(e.sql)
(`a` IN (1, CAST(NULL AS STRING), true))
```

`In` expression can be [evaluated](#) to a boolean value (i.e. `true` or `false`) or the special value `null`.

```

import org.apache.spark.sql.functions.lit
val value = lit(null)
val list = Seq(lit(1))
val in = (value isin (list: _*)).expr

scala> println(in.sql)
(NULL IN (1))

import org.apache.spark.sql.catalyst.InternalRow
val input = InternalRow(1, "hello")

// Case 1: value.eval(input) was null => null
val evaluatedValue = in.eval(input)
assert(evaluatedValue == null)

// Case 2: v = e.eval(input) && ordering.equiv(v, evaluatedValue) => true
val value = lit(1)
val list = Seq(lit(1))
val in = (value isin (list: _*)).expr
val evaluatedValue = in.eval(input)
assert(evaluatedValue.asInstanceOf[Boolean])

// Case 3: e.eval(input) = null and no ordering.equiv(v, evaluatedValue) => null
val value = lit(1)
val list = Seq(lit(null), lit(2))
val in = (value isin (list: _*)).expr
scala> println(in.sql)
(1 IN (NULL, 2))

val evaluatedValue = in.eval(input)
assert(evaluatedValue == null)

// Case 4: false
val value = lit(1)
val list = Seq(0, 1, 2).map(lit)
val in = (value isin (list: _*)).expr
scala> println(in.sql)
(1 IN (0, 1, 2))

val evaluatedValue = in.eval(input)
assert(evaluatedValue.asInstanceOf[Boolean] == false)

```

In takes the following when created:

- Value expression
- Expression list

Note

Expression list must not be null (but can have expressions that can be evaluated to null).

In uses the following **text representation** (i.e. `toString`):

```
[value] IN [list]
```

```
import org.apache.spark.sql.catalyst.expressions.{In, Literal}
import org.apache.spark.sql.{functions => f}
val in = In(value = Literal(1), list = Seq(f.array("1", "2", "3").expr))
scala> println(in)
1 IN (array('1, '2, '3))
```

In has the following [SQL representation](#):

```
([valueSQL] IN ([listSQL]))
```

```
import org.apache.spark.sql.catalyst.expressions.{In, Literal}
import org.apache.spark.sql.{functions => f}
val in = In(value = Literal(1), list = Seq(f.array("1", "2", "3").expr))
scala> println(in.sql)
1 IN (array(`1`, `2`, `3`))
```

In expression is [inSetConvertible](#) when the list contains [Literal](#) expressions only.

```
// FIXME Example 1: inSetConvertible true

// FIXME Example 2: inSetConvertible false
```

In expressions are analyzed using the following rules:

- [ResolveSubquery](#) resolution rule
- [InConversion](#) type coercion rule

In expression has a [custom support](#) in [InMemoryTableScanExec](#) physical operator.

```
// FIXME
// Demo: InMemoryTableScanExec and In expression
// 1. Create an In(a: AttributeReference, list: Seq[Literal]) with the list.nonEmpty
// 2. Use InMemoryTableScanExec.buildFilter partial function to produce the expression
```

Table 1. In's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
ordering	<p>Scala's Ordering instance that represents a strategy for sorting instances of a type.</p> <p>Lazily-instantiated using <code>TypeUtils.getInterpretedOrdering</code> for the data type of the value expression.</p> <p>Used exclusively when <code>In</code> is requested to evaluate a value for a given input row.</p>

Checking Input Data Types — `checkInputDataTypes` Method

```
checkInputDataTypes(): TypeCheckResult
```

Note	<code>checkInputDataTypes</code> is part of the Expression Contract to checks the input data types.
------	---

```
checkInputDataTypes ...FIXME
```

Evaluating Expression — `eval` Method

```
eval(input: InternalRow): Any
```

Note	<code>eval</code> is part of Expression Contract for the interpreted (non-code-generated) expression evaluation , i.e. evaluating a Catalyst expression to a JVM object for a given internal binary row .
------	--

`eval` requests [value](#) expression to [evaluate a value](#) for the `input` [internal row](#).

If the evaluated value is `null`, `eval` gives `null` too.

`eval` takes every [expression](#) in `list` expressions and requests them to evaluate a value for the `input` internal row. If any of the evaluated value is not `null` and equivalent in the `ordering`, `eval` returns `true`.

`eval` records whether any of the expressions in `list` expressions gave `null` value. If no `list` expression led to `true` (per `ordering`), `eval` returns `null` if any `list` expression evaluated to `null` or `false`.

Generating Java Source Code (ExprCode) For Code-Generated Expression Evaluation — doGenCode Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

doGenCode is part of [Expression Contract](#) to generate a Java source code (ExprCode) for code-generated expression evaluation.

doGenCode ...FIXME

```
val in = $"id" isin (1, 2, 3)
val q = spark.range(4).filter(in)
val plan = q.queryExecution.executedPlan

import org.apache.spark.sql.execution.FilterExec
val filterExec = plan.collectFirst { case f: FilterExec => f }.get

import org.apache.spark.sql.catalyst.expressions.In
val inExpr = filterExec.expressions.head.asInstanceOf[In]

import org.apache.spark.sql.execution.WholeStageCodegenExec
val wsce = plan.asInstanceOf[WholeStageCodegenExec]
val (ctx, code) = wsce.doCodeGen

import org.apache.spark.sql.catalyst.expressions.codegen.CodeFormatter
scala> println(CodeFormatter.format(code))
...code omitted

// FIXME Make it work
// I thought I'd reuse ctx to have expression: id#14L evaluated
inExpr.genCode(ctx)
```

Inline Generator Expression

Inline is a unary expression and CollectionGenerator.

Inline is created by inline and inline_outer standard functions.

```
// Query with inline function
val q = spark.range(1)
  .selectExpr("inline(array(struct(1, 'a'), struct(2, 'b')))")
val logicalPlan = q.queryExecution.analyzed
scala> println(logicalPlan.numberedTreeString)
00 Project [col1#61, col2#62]
01 +- Generate inline(array(named_struct(col1, 1, col2, a), named_struct(col1, 2, col2
, b))), false, false, [col1#61, col2#62]
02   +- Range (0, 1, step=1, splits=Some(8))

// Query with inline_outer function
val q = spark.range(1)
  .selectExpr("inline_outer(array(struct(1, 'a'), struct(2, 'b')))")
val logicalPlan = q.queryExecution.analyzed
scala> println(logicalPlan.numberedTreeString)
00 Project [col1#69, col2#70]
01 +- Generate inline(array(named_struct(col1, 1, col2, a), named_struct(col1, 2, col2
, b))), false, true, [col1#69, col2#70]
02   +- Range (0, 1, step=1, splits=Some(8))

import org.apache.spark.sql.catalyst.plans.logical.Generate
// get is safe since there is Generate logical operator
val generator = logicalPlan.collectFirst { case g: Generate => g.generator }.get
import org.apache.spark.sql.catalyst.expressions.Inline
val inline = generator.asInstanceOf[Inline]

// Inline Generator expression is also CollectionGenerator
scala> inline.collectionType.catalogString
res1: String = array<struct<col1:int,col2:string>>
```

InSet Predicate Expression

`InSet` is a [predicate expression](#) (i.e. the result [data type](#) is always [boolean](#)) that is an optimized variant of the [In](#) predicate expression.

InSubquery Expression

`InSubquery` is a [ExecSubqueryExpression](#) that...FIXME

`InSubquery` is [created](#) when...FIXME

updateResult Method

```
updateResult(): Unit
```

Note `updateResult` is part of [ExecSubqueryExpression Contract](#) to...FIXME.

`updateResult` ...FIXME

Creating InSubquery Instance

`InSubquery` takes the following when created:

- Child [expression](#)
- [SubqueryExec](#) physical operator
- Expression ID (as `ExprId`)
- `result` array (default: `null`)
- `updated` flag (default: `false`)

JsonToStructs Unary Expression

`JsonToStructs` is a [unary expression](#) with [timezone](#) support and [CodegenFallback](#).

`JsonToStructs` is [created](#) to represent [from_json](#) function.

```
import org.apache.spark.sql.functions.from_json
val jsonCol = from_json($"json", new StructType())

import org.apache.spark.sql.catalyst.expressions.JsonToStructs
val jsonExpr = jsonCol.expr.asInstanceOf[JsonToStructs]
scala> println(jsonExpr.numberedTreeString)
00 jsontostructs('json, None)
01 +- 'json
```

`JsonToStructs` is a [ExpectInputTypes](#) expression.

Note

`JsonToStructs` uses [JacksonParser](#) in `FAILFAST` mode that simply fails early when a corrupted/malformed record is found (and hence does not support `columnNameOfCorruptRecord` JSON option).

Table 1. `JsonToStructs`'s Properties

Property	Description	
<code>converter</code>	Function that converts <code>Seq[InternalRow]</code> into...FIXME	
<code>nullable</code>	Enabled (i.e. <code>true</code>)	
<code>parser</code>	<code>JacksonParser</code> with <code>rowSchema</code> and <code>JSON options</code>	
	<p>Note</p> <p><code>JSON options</code> are made up of the input <code>options</code> with <code>mode</code> option as <code>FAILFAST</code> and the input <code>time zone</code> as the default time zone.</p>	
<code>rowSchema</code>	<p><code>StructType</code> that...FIXME</p> <ul style="list-style-type: none"> • <code>schema</code> when of type <code>StructType</code> • <code>StructType</code> of the elements in <code>schema</code> when of type <code>ArrayType</code> 	

Creating `JsonToStructs` Instance

`JsonToStructs` takes the following when created:

- [DataType](#)
- [Options](#)
- Child [expression](#)
- Optional time zone ID

`JsonToStructs` initializes the [internal registries and counters](#).

Parsing Table Schema for String Literals — `validateSchemaLiteral` Method

```
validateSchemaLiteral(exp: Expression): StructType
```

`validateSchemaLiteral` requests [CatalystSqlParser](#) to [parseTableSchema](#) for [Literal](#) of [StringType](#).

For any other non- [StringType](#) [types](#), `validateSchemaLiteral` reports a [AnalysisException](#):

```
Expected a string literal instead of [expression]
```

JsonTuple Generator Expression

JsonTuple is...FIXME

ListQuery Subquery Expression

`ListQuery` is a [SubqueryExpression](#) that represents SQL's [IN](#) predicate with a subquery,
e.g. `NOT? IN (' query ')`.

`ListQuery` [cannot be evaluated](#) and produce a value given an internal row.

`ListQuery` is [resolved](#) when:

1. Children are resolved
2. Subquery logical plan is resolved
3. There is at least one [child output attribute](#)

Creating ListQuery Instance

`ListQuery` takes the following when created:

- Subquery [logical plan](#)
- Child [expressions](#)
- Expression ID (as `ExprId` and defaults to a [new ExprId](#))
- Child output [attributes](#)

Literal Leaf Expression

`Literal` is a [leaf expression](#) that is [created](#) to represent a Scala [value](#) of a [specific type](#).

`Literal` is [created](#) when...MEFIXME

Table 1. Literal's Properties

Property	Description
<code>foldable</code>	Enabled (i.e. <code>true</code>)
<code>nullable</code>	Enabled when <code>value</code> is <code>null</code>

Creating Literal Instance — `create` Object Method

```
create(v: Any, dataType: DataType): Literal
```

`create` uses `CatalystTypeConverters` helper object to [convert](#) the input `v` Scala value to a Catalyst rows or types and creates a [Literal](#) (with the Catalyst value and the input `DataType`).

Note	<code>create</code> is used when...FIXME
------	--

Creating Literal Instance

`Literal` takes the following when created:

- Scala value (of type `Any`)
- [DataType](#)

Generating Java Source Code (`ExprCode`) For Code-Generated Expression Evaluation — `doGenCode` Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note	<code>doGenCode</code> is part of Expression Contract to generate a Java source code (<code>ExprCode</code>) for code-generated expression evaluation.
------	--

`doGenCode` ...FIXME

MonotonicallyIncreasingID Nondeterministic Leaf Expression

`MonotonicallyIncreasingID` is a [non-deterministic leaf expression](#) that is the internal representation of the `monotonically_increasing_id` [standard](#) and [SQL](#) functions.

As a `Nondeterministic` expression, `MonotonicallyIncreasingID` requires explicit [initialization](#) (with the current partition index) before [evaluating a value](#).

`MonotonicallyIncreasingID` [generates Java source code \(as ExprCode\)](#) for code-generated expression evaluation.

```
import org.apache.spark.sql.catalyst.expressions.MonotonicallyIncreasingID
val monotonicallyIncreasingID = MonotonicallyIncreasingID()

import org.apache.spark.sql.catalyst.expressions.codegen.{CodegenContext, ExprCode}
val ctx = new CodegenContext
// doGenCode is used when Expression.genCode is executed
val ExprCode(code, _, _) = monotonicallyIncreasingID.genCode(ctx)

// Helper methods
def trim(code: String): String = {
  code.trim.split("\n").map(_.trim).filter(line => line.nonEmpty).mkString("\n")
}
def prettyPrint(code: String) = println(trim(code))
// END: Helper methods

scala> println(trim(code))
final long value_0 = partitionMask + count_0;
count_0++;
```

`MonotonicallyIncreasingID` uses [LongType](#) as the [data type](#) of the result of evaluating itself.

`MonotonicallyIncreasingID` is never [nullable](#).

`MonotonicallyIncreasingID` uses [monotonically_increasing_id](#) for the [user-facing name](#).

`MonotonicallyIncreasingID` uses [monotonically_increasing_id\(\)](#) for the [SQL representation](#).

`MonotonicallyIncreasingID` is [created](#) when `monotonically_increasing_id` standard function is used in a structured query.

`MonotonicallyIncreasingID` is [registered](#) as `monotonically_increasing_id` SQL function.

`MonotonicallyIncreasingID` takes no input parameters when created.

Table 1. MonotonicallyIncreasingID's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
count	<p>Number of <code>evalInternal</code> calls, i.e. the number of rows for which <code>MonotonicallyIncreasingID</code> was evaluated</p> <p>Initialized when <code>MonotonicallyIncreasingID</code> is requested to <code>initialize</code> and used to <code>evaluate a value</code>.</p>
partitionMask	<p>Current partition index shifted 33 bits left</p> <p>Initialized when <code>MonotonicallyIncreasingID</code> is requested to <code>initialize</code> and used to <code>evaluate a value</code>.</p>

Generating Java Source Code (ExprCode) For Code-Generated Expression Evaluation — doGenCode Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

`doGenCode` is part of [Expression Contract](#) to generate a Java source code (`ExprCode`) for code-generated expression evaluation.

`doGenCode` requests the `CodegenContext` to [add a mutable state](#) as `count` `name` and `long` Java type.

`doGenCode` requests the `CodegenContext` to [add an immutable state \(unless exists already\)](#) as `partitionMask` `name` and `long` Java type.

`doGenCode` requests the `CodegenContext` to [addPartitionInitializationStatement](#) with `[countTerm] = 0L;` statement.

`doGenCode` requests the `CodegenContext` to [addPartitionInitializationStatement](#) with `[partitionMaskTerm] = ((long) partitionIndex) << 33;` statement.

In the end, `doGenCode` returns the input `ExprCode` with the `code` as follows and `isNull` property disabled (`false`):

```
final [dataType] [value] = [partitionMaskTerm] + [countTerm];
[value]++;
```

Initializing Nondeterministic Expression — initializeInternal Method

```
initializeInternal(input: InternalRow): Long
```

Note

`initializeInternal` is part of [Nondeterministic Contract](#) to initialize a Nondeterministic expression.

`initializeInternal` simply sets the `count` to `0` and the `partitionMask` to `partitionIndex.toLong << 33`.

```
val partitionIndex = 1
val partitionMask = partitionIndex.toLong << 33
scala> println(partitionMask.toBinaryString)
1000000000000000000000000000000000000000000000000000000000000000
```

Evaluating Nondeterministic Expression — `evalInternal` Method

```
evalInternal(input: InternalRow): Long
```

Note

`evalInternal` is part of [Nondeterministic Contract](#) to evaluate the value of a Nondeterministic expression.

`evalInternal` remembers the current value of the `count` and increments it.

In the end, `evalInternal` returns the sum of the current value of the `partitionMask` and the remembered value of the `count`.

Murmur3Hash

Murmur3Hash is...FIXME

NamedExpression Contract — Catalyst Expressions with Name, ID and Qualifier

`NamedExpression` is a [contract](#) of Catalyst expressions that have a [name](#), [exprId](#), and optional [qualifier](#).

```
package org.apache.spark.sql.catalyst.expressions

trait NamedExpression extends Expression {
    // only required methods that have no implementation
    def exprId: ExprId
    def name: String
    def newInstance(): NamedExpression
    def qualifier: Option[String]
    def toAttribute: Attribute
}
```

Table 1. NamedExpression Contract

Method	Description
<code>exprId</code>	Used when...FIXME
<code>name</code>	Used when...FIXME
<code>qualifier</code>	Used when...FIXME
<code>toAttribute</code>	

Creating ExprId — `newExprId` Object Method

```
newExprId: ExprId
```

```
newExprId ...FIXME
```

Note	<code>newExprId</code> is used when...FIXME
------	---

Nondeterministic Expression Contract

`Nondeterministic` is a [contract](#) for [Catalyst expressions](#) that are non-deterministic and non-foldable.

`Nondeterministic` expressions require explicit [initialization](#) (with the current partition index) before [evaluating a value](#).

```
package org.apache.spark.sql.catalyst.expressions

trait Nondeterministic extends Expression {
    // only required methods that have no implementation
    protected def initializeInternal(partitionIndex: Int): Unit
    protected def evalInternal(input: InternalRow): Any
}
```

Table 1. Nondeterministic Contract

Method	Description
<code>initializeInternal</code>	Initializing the <code>Nondeterministic</code> expression Used exclusively when <code>Nondeterministic</code> expression is requested to initialize
<code>evalInternal</code>	Evaluating the <code>Nondeterministic</code> expression Used exclusively when <code>Nondeterministic</code> expression is requested to evaluate a value
Note	<code>Nondeterministic</code> expressions are the target of <code>PullOutNondeterministic</code> logical plan rule.

Table 2. Nondeterministic Expressions

Expression	Description
CurrentBatchTimestamp	
InputFileBlockLength	
InputFileBlockStart	
InputFileName	
MonotonicallyIncreasingID	
NondeterministicExpression	
Rand	
Randn	
RDG	
SparkPartitionID	

Table 3. Nondeterministic's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
deterministic	Always turned off (i.e. <code>false</code>)
foldable	Always turned off (i.e. <code>false</code>)
initialized	Controls whether a <code>Nondeterministic</code> expression has been initialized before evaluation . Turned off by default.

Initializing Expression — `initialize` Method

```
initialize(partitionIndex: Int): Unit
```

Internally, `initialize` [initializes](#) itself (with the input partition index) and turns the internal `initialized` flag on.

Note	<code>initialize</code> is used exclusively when <code>InterpretedProjection</code> and <code>InterpretedMutableProjection</code> are requested to <code>initialize</code> themselves.
------	--

Evaluating Expression — `eval` Method

```
eval(input: InternalRow): Any
```

Note

`eval` is part of [Expression Contract](#) for the **interpreted (non-code-generated) expression evaluation**, i.e. evaluating a Catalyst expression to a JVM object for a given [internal binary row](#).

`eval` is just a wrapper of `evalInternal` that makes sure that `initialize` has already been executed (and so the expression is initialized).

Internally, `eval` makes sure that the expression was [initialized](#) and calls `evalInternal`.

`eval` reports a `IllegalArgumentException` exception when the internal [initialized](#) flag is off, i.e. `initialize` has not yet been executed.

```
requirement failed: Nondeterministic expression [name] should be initialized before eval.
```

OffsetWindowFunction Contract — Unevaluable Window Function Expressions

`OffsetWindowFunction` is the [base](#) of window function expressions that are [unevaluable](#) and [ImplicitCastInputTypes](#).

Note

An [unevaluable expression](#) cannot be evaluated to produce a value (neither in [interpreted](#) nor [code-generated](#) expression evaluations) and has to be resolved (replaced) to some other expressions or logical operators at [analysis](#) or [optimization](#) phases or they fail analysis.

```
package org.apache.spark.sql.catalyst.expressions

abstract class OffsetWindowFunction ... {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    val default: Expression
    val direction: SortDirection
    val input: Expression
    val offset: Expression
}
```

Table 1. (Subset of) OffsetWindowFunction Contract

Property	Description
<code>default</code>	
<code>direction</code>	
<code>input</code>	
<code>offset</code>	

`OffsetWindowFunction` uses the [input](#), [offset](#) and [default](#) expressions as the [children](#).

`OffsetWindowFunction` is not [foldable](#).

`OffsetWindowFunction` is [nullable](#) when the [default](#) is not defined or the [default](#) or the [input](#) expressions are.

When requested for the [dataType](#), `OffsetWindowFunction` simply requests the [input](#) expression for the data type.

When requested for the [inputTypes](#), `OffsetWindowFunction` returns the [AnyDataType](#), [IntegerType](#) with the data type of the [input](#) expression and the [NullType](#).

`OffsetWindowFunction` uses the following **text representation** (i.e. `toString`):

```
[prettyName]([input], [offset], [default])
```

Table 2. OffsetWindowFunctions (Direct Implementations)

OffsetWindowFunction	Description
Lag	
Lead	

frame Lazy Property

```
frame: WindowFrame
```

Note	<code>frame</code> is part of the WindowFunction Contract to define the <code>WindowFrame</code> for function expression execution.
------	---

`frame` ...FIXME

Verifying Input Data Types — `checkInputDataTypes` Method

```
checkInputDataTypes(): TypeCheckResult
```

Note	<code>checkInputDataTypes</code> is part of the Expression Contract to verify (check the correctness of) the input data types.
------	--

`checkInputDataTypes` ...FIXME

ParseToDate Expression

`ParseToDate` is a [RuntimeReplaceable](#) expression that [represents](#) the `to_date` function (in logical query plans).

```
// DEMO to_date(e: Column): Column  
// DEMO to_date(e: Column, fmt: String): Column
```

As a `RuntimeReplaceable` expression, `ParseToDate` is replaced by [Catalyst Optimizer](#) with the [child](#) expression:

- `Cast(left, DateType)` for `to_date(e: Column): Column` function
- `Cast(Cast(UnixTimestamp(left, format), TimestampType), DateType)` for `to_date(e: Column, fmt: String): Column` function

```
// FIXME DEMO Conversion to `Cast(left, DateType)`  
// FIXME DEMO Conversion to `Cast(Cast(UnixTimestamp(left, format), TimestampType), Da-  
teType)`
```

Creating ParseToDate Instance

`ParseToDate` takes the following when created:

- Left [expression](#)
- `format` [expression](#)
- Child [expression](#)

ParseToTimestamp Expression

`ParseToTimestamp` is a [RuntimeReplaceable](#) expression that represents the `to_timestamp` function (in logical query plans).

```
// DEMO to_timestamp(s: Column): Column
import java.sql.Timestamp
import java.time.LocalDateTime
val times = Seq(Timestamp.valueOf(LocalDateTime.of(2018, 5, 30, 0, 0, 0)).toString).toDF("time")
scala> times.printSchema
root
| -- time: string (nullable = true)

import org.apache.spark.sql.functions.to_timestamp
val q = times.select(to_timestamp($"time") as "ts")
scala> q.printSchema
root
| -- ts: timestamp (nullable = true)

val plan = q.queryExecution.analyzed
scala> println(plan.numberedTreeString)
00 Project [to_timestamp('time, None) AS ts#29]
01 +- Project [value#16 AS time#18]
02   +- LocalRelation [value#16]

import org.apache.spark.sql.catalyst.expressions.ParseToTimestamp
val ptt = plan.expressions.head.children.head.asInstanceOf[ParseToTimestamp]
scala> println(ptt.numberedTreeString)
00 to_timestamp('time, None)
01 +- cast(time#18 as timestamp)
02   +- time#18: string

// FIXME DEMO to_timestamp(s: Column, fmt: String): Column
```

As a `RuntimeReplaceable` expression, `ParseToTimestamp` is replaced by [Catalyst Optimizer](#) with the `child` expression:

- `Cast(left, TimestampType)` for `to_timestamp(s: Column): Column` function
- `Cast(UnixTimestamp(left, format), TimestampType)` for `to_timestamp(s: Column, fmt: String): Column` function

```
// FIXME DEMO Conversion to `Cast(left, TimestampType)`
// FIXME DEMO Conversion to `Cast(UnixTimestamp(left, format), TimestampType)`
```

Creating ParseToTimestamp Instance

`ParseToTimestamp` takes the following when created:

- Left `expression`
- `format` `expression`
- Child `expression`

PlanExpression Contract for Expressions with Query Plans

`PlanExpression` is the [contract](#) for [Catalyst expressions](#) that contain a [QueryPlan](#).

```
package org.apache.spark.sql.catalyst.expressions

abstract class PlanExpression[T <: QueryPlan[_]] extends Expression {
    // only required methods that have no implementation
    // the others follow
    def exprId: ExprId
    def plan: T
    def withNewPlan(plan: T): PlanExpression[T]
}
```

Table 1. PlanExpression Contract

Method	Description
<code>exprId</code>	Used when...FIXME
<code>plan</code>	Used when...FIXME
<code>withNewPlan</code>	Used when...FIXME

Table 2. PlanExpressions

PlanExpression	Description
ExecSubqueryExpression	
SubqueryExpression	

PrettyAttribute

PrettyAttribute is...FIXME

RankLike Contract

RankLike is...FIXME

ResolvedStar

ResolvedStar is...FIXME

RowNumberLike Contract

RowNumberLike is...FIXME

RuntimeReplaceable Contract — Replaceable SQL Expressions

`RuntimeReplaceable` is the [marker contract](#) for [unary expressions](#) that are replaced by [Catalyst Optimizer](#) with their child expression (that can then be evaluated).

Note

Catalyst Optimizer uses [ReplaceExpressions](#) logical optimization to replace `RuntimeReplaceable` expressions.

`RuntimeReplaceable` contract allows for **expression aliases**, i.e. expressions that are fairly complex in the inside than on the outside, and is used to provide compatibility with other SQL databases by supporting SQL functions with their more complex Catalyst expressions (that are already supported by Spark SQL).

Note

`RuntimeReplaceables` are tied up to their SQL functions in [FunctionRegistry](#).

`RuntimeReplaceable` expressions [cannot be evaluated](#) (i.e. produce a value given an internal row) and therefore have to be replaced in the [query execution pipeline](#).

```
package org.apache.spark.sql.catalyst.expressions

trait RuntimeReplaceable extends UnaryExpression with Unevaluable {
    // as a marker contract it only marks a class
    // no methods are required
}
```

Note

To make sure the `explain` plan and expression SQL works correctly, a `RuntimeReplaceable` implementation should override [flatArguments](#) and [sql](#) methods.

Table 1. RuntimeReplaceables

RuntimeReplaceable	Standard Function	SQL Function
IfNull		ifnull
Left		left
NullIf		nullif
Nvl		nvl
Nvl2		nvl2
ParseToDate	to_date	to_date
ParseToTimestamp	to_timestamp	to_timestamp
Right		right

ScalarSubquery (SubqueryExpression) Expression

`ScalarSubquery` is a [SubqueryExpression](#) that returns a single row and a single column only.

`ScalarSubquery` represents a structured query that can be used as a "column".

Important	Spark SQL uses the name of <code>ScalarSubquery</code> twice to represent a SubqueryExpression (this page) and an ExecSubqueryExpression . You've been warned.
-----------	--

`ScalarSubquery` is [created](#) exclusively when `AstBuilder` is requested to [parse a subquery expression](#).

```
// FIXME DEMO

// Borrowed from ExpressionParserSuite.scala
// ScalarSubquery(table("tbl").select('max.function('val))) > 'current)
val sql = "(select max(val) from tbl) > current"

// 'a === ScalarSubquery(table("s").select('b))
val sql = "a = (select b from s)"

// Borrowed from PlanParserSuite.scala
// table("t").select(ScalarSubquery(table("s").select('max.function('b))).as("ss"))
val sql = "select (select max(b) from s) ss from t"

// table("t").where('a === ScalarSubquery(table("s").select('b))).select(star())
val sql = "select * from t where a = (select b from s)"

// table("t").groupBy('g)(`g).where('a > ScalarSubquery(table("s").select('b)))
val sql = "select g from t group by g having a > (select b from s)"
```

Creating ScalarSubquery Instance

`ScalarSubquery` takes the following when created:

- Subquery [logical plan](#)
- Child [expressions](#) (default: no children)
- Expression ID (as `ExprId` and defaults to a [new ExprId](#))

ScalarSubquery (ExecSubqueryExpression) Expression

`ScalarSubquery` is an [ExecSubqueryExpression](#) that can give exactly one value (i.e. the value of executing [SubqueryExec](#) subquery that can result in a single row and a single column or `null` if no row were computed).

Important

Spark SQL uses the name of `ScalarSubquery` twice to represent an [ExecSubqueryExpression](#) (this page) and a [SubqueryExpression](#). It is confusing and you should *not* be anymore.

`ScalarSubquery` is [created](#) exclusively when `PlanSubqueries` physical optimization is [executed](#) (and plans a [ScalarSubquery](#) expression).

```
// FIXME DEMO
import org.apache.spark.sql.execution.PlanSubqueries
val spark = ...
val planSubqueries = PlanSubqueries(spark)
val plan = ...
val executedPlan = planSubqueries(plan)
```

`ScalarSubquery` expression [cannot be evaluated](#), i.e. produce a value given an internal row.

`ScalarSubquery` uses...FIXME...for the [data type](#).

Table 1. ScalarSubquery's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>result</code>	The value of the single column in a single row after collecting the rows from executing the subquery plan or <code>null</code> if no rows were collected.
<code>updated</code>	Flag that says whether <code>ScalarSubquery</code> was updated with collected result of executing the subquery plan .

Creating ScalarSubquery Instance

`ScalarSubquery` takes the following when created:

- [SubqueryExec](#) plan
- Expression ID (as `ExprId`)

Updating ScalarSubquery With Collected Result — `updateResult` Method

```
updateResult(): Unit
```

Note

`updateResult` is part of [ExecSubqueryExpression Contract](#) to fill an Catalyst expression with a collected result from executing a subquery plan.

`updateResult` requests [SubqueryExec](#) physical plan to [execute and collect internal rows](#).

`updateResult` sets `result` to the value of the only column of the single row or `null` if no row were collected.

In the end, `updateResult` marks the `ScalarSubquery` instance as [updated](#).

`updateResult` reports a `RuntimeException` when there are more than 1 rows in the result.

more than one row returned by a subquery used as an expression:

[plan]

`updateResult` reports an `AssertionError` when the number of fields is not exactly 1.

Expects 1 field, but got [numFields] something went wrong in analysis

Evaluating Expression — `eval` Method

```
eval(input: InternalRow): Any
```

Note

`eval` is part of [Expression Contract](#) for the **interpreted (non-code-generated) expression evaluation**, i.e. evaluating a Catalyst expression to a JVM object for a given [internal binary row](#).

`eval` simply returns `result` value.

`eval` reports an `IllegalArgumentException` if the `ScalarSubquery` expression has not been [updated](#) yet.

Generating Java Source Code (`ExprCode`) For Code-Generated Expression Evaluation — `doGenCode` Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

`doGenCode` is part of [Expression Contract](#) to generate a Java source code (`ExprCode`) for code-generated expression evaluation.

`doGenCode` first makes sure that the `updated` flag is on (`true`). If not, `doGenCode` throws an `IllegalArgumentException` exception with the following message:

```
requirement failed: [this] has not finished
```

`doGenCode` then creates a [Literal](#) (for the `result` and the `dataType`) and simply requests it to generate a Java source code.

ScalaUDF — Catalyst Expression to Manage Lifecycle of User-Defined Function

`ScalaUDF` is a [Catalyst expression](#) to manage the lifecycle of a [user-defined function](#) (and hook it in to Spark SQL's Catalyst execution path).

`ScalaUDF` is a `ImplicitCastInputTypes` and `UserDefinedExpression`.

`ScalaUDF` has [no representation in SQL](#).

`ScalaUDF` is created when:

- `UserDefinedFunction` is [executed](#)
- `UDFRegistration` is requested to [register a Scala function as a user-defined function](#) (in `FunctionRegistry`)

```
val lengthUDF = udf { s: String => s.length }.withName("lengthUDF")
val c = lengthUDF($"name")
scala> println(c.expr.treeString)
UDF:lengthUDF('name)
+- 'name

import org.apache.spark.sql.catalyst.expressions.ScalaUDF
val scalaUDF = c.expr.asInstanceOf[ScalaUDF]
```

Note

[Spark SQL Analyzer](#) uses `HandleNullInputsForUDF` logical evaluation rule to...
FIXME

```
// Defining a zero-argument UDF
val myUDF = udf { () => "Hello World" }

// "Execute" the UDF
// Attach it to an "execution environment", i.e. a Dataset
// by specifying zero columns to execute on (since the UDF is no-arg)
import org.apache.spark.sql.catalyst.expressions.ScalaUDF
val scalaUDF = myUDF().expr.asInstanceOf[ScalaUDF]

scala> scalaUDF.resolved
res1: Boolean = true

// Execute the UDF (on every row in a Dataset)
// We simulate it relying on the EmptyRow that is the default InternalRow of eval
scala> scalaUDF.eval()
res2: Any = Hello World
```

```

// Defining a UDF of one input parameter
val hello = udf { s: String => s"Hello $s" }

// Binding the hello UDF to a column name
import org.apache.spark.sql.catalyst.expressions.ScalaUDF
val helloScalaUDF = hello($"name").expr.asInstanceOf[ScalaUDF]

scala> helloScalaUDF.resolved
res3: Boolean = false

// Resolve helloScalaUDF, i.e. the only `name` column reference

scala> helloScalaUDF.children
res4: Seq[org.apache.spark.sql.catalyst.expressions.Expression] = ArrayBuffer('name)

// The column is free (i.e. not bound to a Dataset)
// Define a Dataset that becomes the rows for the UDF
val names = Seq("Jacek", "Agata").toDF("name")
scala> println(names.queryExecution.analyzed.numberedTreeString)
00 Project [value#1 AS name#3]
01 +- LocalRelation [value#1]

// Resolve the references using the Dataset
val plan = names.queryExecution.analyzed
val resolver = spark.sessionState.analyzer.resolver
import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
val resolvedUDF = helloScalaUDF.transformUp { case a @ UnresolvedAttribute(names) =>
  // we're in controlled environment
  // so get is safe
  plan.resolve(names, resolver).get
}

scala> resolvedUDF.resolved
res6: Boolean = true

scala> println(resolvedUDF.numberedTreeString)
00 UDF(name#3)
01 +- name#3: string

import org.apache.spark.sql.catalyst.expressions.BindReferences
val attrs = names.queryExecution.sparkPlan.output
val boundUDF = BindReferences.bindReference(resolvedUDF, attrs)

// Create an internal binary row, i.e. InternalRow
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val stringEncoder = ExpressionEncoder[String]
val row = stringEncoder.toRow("world")

// YAY! It works!
scala> boundUDF.eval(row)
res8: Any = Hello world

// Just to show the regular execution path

```

```
// i.e. how to execute a UDF in a context of a Dataset
val q = names.select(hello($"name"))
scala> q.show
+-----+
| UDF(name)|
+-----+
|Hello Jacek|
|Hello Agata|
+-----+
```

`ScalaUDF` is deterministic when the given `udfDeterministic` flag is enabled (`true`) and all the `children expressions` are deterministic.

Generating Java Source Code (`ExprCode`) For Code-Generated Expression Evaluation — `doGenCode` Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note	<code>doGenCode</code> is part of Expression Contract to generate a Java source code (<code>ExprCode</code>) for code-generated expression evaluation.
------	--

`doGenCode` ...FIXME

Evaluating Expression — `eval` Method

```
eval(input: InternalRow): Any
```

Note	<code>eval</code> is part of Expression Contract for the interpreted (non-code-generated) expression evaluation , i.e. evaluating a Catalyst expression to a JVM object for a given internal binary row .
------	--

`eval` executes the [Scala function](#) on the input [internal row](#).

Creating ScalaUDF Instance

`ScalaUDF` takes the following when created:

- A Scala function (as Scala's `AnyRef`)
- Output [data type](#)
- Child [Catalyst expressions](#)
- Input [data types](#) (default: `Nil`)

- `Optional name (default: None)`
- `nullable flag (default: true)`
- `udfDeterministic flag (default: true)`

`ScalaUDF` initializes the [internal registries and counters](#).

ScalaUDAF — Catalyst Expression Adapter for UserDefinedAggregateFunction

`ScalaUDAF` is a [Catalyst expression](#) adapter to manage the lifecycle of [UserDefinedAggregateFunction](#) and hook it in Spark SQL's Catalyst execution path.

`ScalaUDAF` is [created](#) when:

- `UserDefinedAggregateFunction` creates a `Column` for a user-defined aggregate function using `all` and `distinct` values (to use the UDAF in [Dataset operators](#))
- `UDFRegistration` is requested to register a user-defined aggregate function (to use the UDAF in [SQL mode](#))

`ScalaUDAF` is a [ImperativeAggregate](#).

Table 1. ScalaUDAF's ImperativeAggregate Methods

Method Name	Behaviour
<code>initialize</code>	Requests UserDefinedAggregateFunction to initialize
<code>merge</code>	Requests UserDefinedAggregateFunction to merge
<code>update</code>	Requests UserDefinedAggregateFunction to update

When evaluated, `ScalaUDAF` ...FIXME

`ScalaUDAF` has [no representation in SQL](#).

Table 2. ScalaUDAF's Properties

Name	Description
aggBufferAttributes	AttributeReferences of aggBufferSchema
aggBufferSchema	bufferSchema of UserDefinedAggregateFunction
dataType	DataType of UserDefinedAggregateFunction
deterministic	deterministic of UserDefinedAggregateFunction
inputAggBufferAttributes	Copy of aggBufferAttributes
inputTypes	Data types from inputSchema of UserDefinedAggregateFunction
nullable	Always enabled (i.e. true)

Table 3. ScalaUDAF's Internal Registries and Counters

Name	Description
inputAggregateBuffer	Used when...FIXME
inputProjection	Used when...FIXME
inputToScalaConverters	Used when...FIXME
mutableAggregateBuffer	Used when...FIXME

Creating ScalaUDAF Instance

ScalaUDAF takes the following when created:

- Children Catalyst expressions
- UserDefinedAggregateFunction
- mutableAggBufferOffset (starting with 0)
- inputAggBufferOffset (starting with 0)

ScalaUDAF initializes the internal registries and counters.

initialize Method

```
initialize(buffer: InternalRow): Unit
```

`initialize` sets the input `buffer` `internal binary row` as `underlyingBuffer` of `MutableAggregationBufferImpl` and requests the `UserDefinedAggregateFunction` to `initialize` (with the `MutableAggregationBufferImpl`).

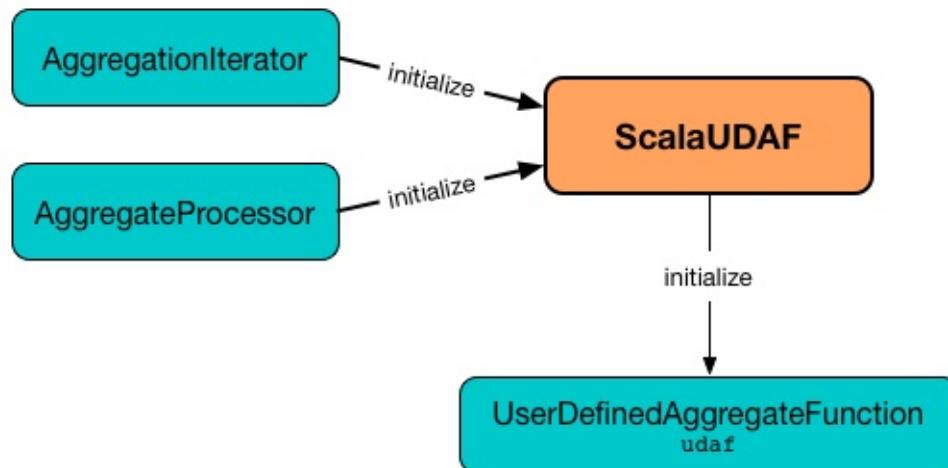


Figure 1. ScalaUDAF initializes UserDefinedAggregateFunction

Note

`initialize` is part of [ImperativeAggregate Contract](#).

update Method

```
update(mutableAggBuffer: InternalRow, inputRow: InternalRow): Unit
```

`update` sets the input `buffer` `internal binary row` as `underlyingBuffer` of `MutableAggregationBufferImpl` and requests the `UserDefinedAggregateFunction` to `update`.

Note

`update` uses `inputProjection` on the input `input` and converts it using `inputToScalaConverters`.

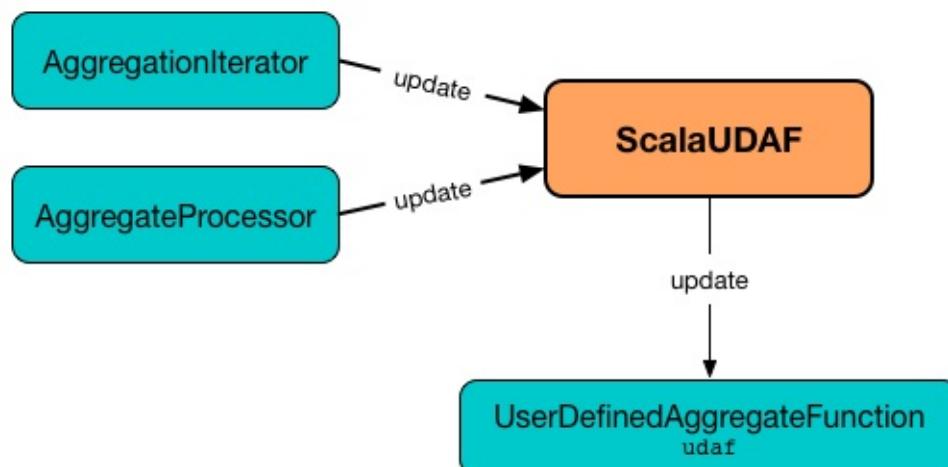


Figure 2. ScalaUDAF updates UserDefinedAggregateFunction

Note	<code>update</code> is part of ImperativeAggregate Contract .
------	---

merge Method

```
merge(buffer1: InternalRow, buffer2: InternalRow): Unit
```

`merge` first sets:

- `underlyingBuffer` of [MutableAggregationBufferImpl](#) to the input `buffer1`
- `underlyingInputBuffer` of [InputAggregationBuffer](#) to the input `buffer2`

`merge` then requests the [UserDefinedAggregateFunction](#) to `merge` (passing in the [MutableAggregationBufferImpl](#) and [InputAggregationBuffer](#)).

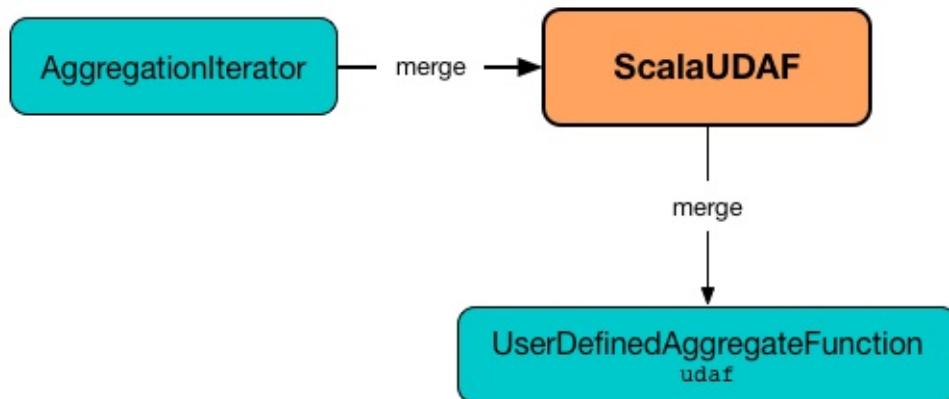


Figure 3. ScalaUDAF requests UserDefinedAggregateFunction to merge

Note	<code>merge</code> is part of ImperativeAggregate Contract .
------	--

SimpleTypedAggregateExpression

`SimpleTypedAggregateExpression` is...FIXME

`SimpleTypedAggregateExpression` is [created](#) when...FIXME

Table 1. SimpleTypedAggregateExpression's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>evaluateExpression</code>	Expression
<code>resultObjToRow</code>	UnsafeProjection

Creating SimpleTypedAggregateExpression Instance

`SimpleTypedAggregateExpression` takes the following when created:

- [Aggregator](#)
- Optional input deserializer [expression](#)
- Optional Java class for the input
- Optional [schema](#) for the input
- Buffer serializer (as a collection of [named expressions](#))
- Buffer deserializer [expression](#)
- Output serializer (as a collection of [expressions](#))
- [DataType](#)
- `nullable` flag

SizeBasedWindowFunction Contract — Declarative Window Aggregate Functions with Window Size

`SizeBasedWindowFunction` is the [extension](#) of the [AggregateWindowFunction Contract](#) for [window functions](#) that require the [size of the current window](#) for calculation.

```
package org.apache.spark.sql.catalyst.expressions

trait SizeBasedWindowFunction extends AggregateWindowFunction {
    // No required properties (vals and methods) that have no implementation
}
```

Table 1. `SizeBasedWindowFunction` Contract

Property	Description
<code>n</code>	Size of the current window as a AttributeReference expression with <code>window_partition_size</code> name, IntegerType data type and not nullable

Table 2. `SizeBasedWindowFunctions` (Direct Implementations)

SizeBasedWindowFunction	Description
<code>CumeDist</code>	Window function expression for <code>cume_dist</code> standard function (Dataset API) and <code>cume_dist</code> SQL function
<code>NTile</code>	
<code>PercentRank</code>	

SortOrder Unevaluable Unary Expression

`SortOrder` is a [unary expression](#) that represents the following operators in a logical plan:

- `AstBuilder` is requested to [parse ORDER BY or SORT BY sort specifications](#)
- `Column.asc`, `Column.asc_nulls_first`, `Column.asc_nulls_last`, `Column.desc`, `Column.desc_nulls_first`, and `Column.desc_nulls_last` operators are used

`SortOrder` is used to specify the [output data ordering requirements](#) of a physical operator.

`SortOrder` is an [unevaluable expression](#) and cannot be evaluated (i.e. produce a value given an internal row).

Note	An unevaluable expression cannot be evaluated to produce a value (neither in interpreted nor code-generated expression evaluations) and has to be resolved (replaced) to some other expressions or logical operators at analysis or optimization phases or they fail analysis.
------	--

`SortOrder` is never [foldable](#) (as an unevaluable expression with no evaluation).

Tip	Use <code>asc</code> , <code>asc_nullsLast</code> , <code>desc</code> or <code>desc_nullsFirst</code> operators from the Catalyst DSL to create a <code>SortOrder</code> expression, e.g. for testing or Spark SQL internals exploration.
-----	---

Note	<code>Dataset.repartitionByRange</code> , <code>Dataset.sortWithinPartitions</code> , <code>Dataset.sort</code> and <code>WindowSpec.orderBy</code> default to Ascending sort direction.
------	--

Creating SortOrder Instance — `apply` Factory Method

```
apply(
  child: Expression,
  direction: SortDirection,
  sameOrderExpressions: Set[Expression] = Set.empty): SortOrder
```

`apply` is a convenience method to create a `SortOrder` with the `defaultNullOrdering` of the `SortDirection`.

Note	<code>apply</code> is used exclusively in window function.
------	--

Catalyst DSL — `asc`, `asc_nullsLast`, `desc` and `desc_nullsFirst` Operators

```
asc: SortOrder
asc_nullsLast: SortOrder
desc: SortOrder
desc_nullsFirst: SortOrder
```

`asc`, `asc_nullsLast`, `desc` and `desc_nullsFirst` [create](#) a `SortOrder` expression with the Ascending Or Descending sort direction, respectively.

```
import org.apache.spark.sql.catalyst.dsl.expressions._
val sortNullsLast = 'id.asc_nullsLast
scala> println(sortNullsLast.sql)
`id` ASC NULLS LAST
```

Creating SortOrder Instance

`SortOrder` takes the following when created:

- Child [expression](#)
- [SortDirection](#)
- `NullOrdering`
- "Same Order" [expressions](#)

SortDirection Contract

`SortDirection` is the [base](#) of sort directions.

Table 1. SortDirection Contract

Method	Description
<code>defaultNullOrdering</code>	<code>defaultNullOrdering: NullOrdering</code> Used when...FIXME
<code>sql</code>	<code>sql: String</code> Used when...FIXME

Ascending and Descending Sort Directions

There are two [sorting directions](#) available, i.e. `Ascending` and `Descending`.

Stack Generator Expression

Stack is...FIXME

Generating Java Source Code (ExprCode) For Code-Generated Expression Evaluation — doGenCode Method

```
doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
```

Note

doGenCode is part of [Expression Contract](#) to generate a Java source code (ExprCode) for code-generated expression evaluation.

doGenCode ...FIXME

Star Expression Contract

`Star` is a [contract](#) of [leaf](#) and [named expressions](#) that...FIXME

```
package org.apache.spark.sql.catalyst.analysis

abstract class Star extends LeafExpression with NamedExpression {
    // only required methods that have no implementation
    def expand(input: LogicalPlan, resolver: Resolver): Seq[NamedExpression]
}
```

Table 1. Star Contract

Method	Description
<code>expand</code>	Used exclusively when <code>ResolveReferences</code> logical resolution rule is requested to expand <code>Star</code> expressions in the following logical operators: <ul style="list-style-type: none"> • ScriptTransformation • Project and Aggregate

Table 2. Stars

Star	Description
ResolvedStar	
UnresolvedRegex	
UnresolvedStar	

StaticInvoke Non-SQL Expression

`StaticInvoke` is an [expression](#) with [no SQL representation](#) that represents a static method call in Scala or Java.

`StaticInvoke` supports [Java code generation](#) (aka *whole-stage codegen*) to evaluate itself.

`StaticInvoke` is [created](#) when:

- `ScalaReflection` is requested for the [deserializer](#) or [serializer](#) for a Scala type
- `RowEncoder` is requested for `deserializerFor` or [serializer](#) for a Scala type
- `JavaTypeInference` is requested for `deserializerFor` or `serializerFor`

```
import org.apache.spark.sql.types.StructType
val schema = new StructType()
  .add($"id".long.copy(nullable = false))
  .add($"name".string.copy(nullable = false))

import org.apache.spark.sql.catalyst.encoders.RowEncoder
val encoder = RowEncoder(schema)
scala> println(encoder.serializer(0).numberedTreeString)
00 validateexternaltype(getexternalrowfield(assertnotnull(input[0, org.apache.spark.sql.Row, true]), 0, id), LongType) AS id#1640L
01 +- validateexternaltype(getexternalrowfield(assertnotnull(input[0, org.apache.spark.sql.Row, true]), 0, id), LongType)
02     +- getexternalrowfield(assertnotnull(input[0, org.apache.spark.sql.Row, true]), 0, id)
03         +- assertnotnull(input[0, org.apache.spark.sql.Row, true])
04             +- input[0, org.apache.spark.sql.Row, true]
```

Note `StaticInvoke` is similar to `CallMethodViaReflection` expression.

Creating StaticInvoke Instance

`StaticInvoke` takes the following when created:

- Target object of the static call
- [Data type](#) of the return value of the [method](#)
- Name of the method to call on the [static object](#)
- Optional [expressions](#) to pass as input arguments to the [function](#)

- Flag to control whether to propagate `nulls` or not (enabled by default). If any of the arguments is `null`, `null` is returned instead of calling the [function](#)

SubqueryExpression Contract — Expressions With Logical Query Plans

`SubqueryExpression` is the [contract](#) for [expressions with logical query plans](#) (i.e. `PlanExpression[LogicalPlan]`).

```
package org.apache.spark.sql.catalyst.expressions

abstract class SubqueryExpression(
    plan: LogicalPlan,
    children: Seq[Expression],
    exprId: ExprId) extends PlanExpression[LogicalPlan] {
    // only required methods that have no implementation
    // the others follow
    override def withNewPlan(plan: LogicalPlan): SubqueryExpression
}
```

Table 1. (Subset of) SubqueryExpression Contract

Method	Description
<code>withNewPlan</code>	<p>Used when:</p> <ul style="list-style-type: none"> • <code>CTESubstitution</code> substitution analyzer rule is requested to <code>substituteCTE</code> • <code>ResolveReferences</code> logical resolution rule is requested to <code>dedupRight</code> and <code>dedupOuterReferencesInSubquery</code> • <code>ResolveSubquery</code> logical resolution rule is requested to <code>resolveSubQuery</code> • <code>UpdateOuterReferences</code> logical rule is <code>executed</code> • <code>ResolveTimeZone</code> logical resolution rule is <code>executed</code> • <code>SubqueryExpression</code> is requested for a canonicalized version • <code>OptimizeSubqueries</code> logical query optimization is <code>executed</code> • <code>CacheManager</code> is requested to replace logical query segments with cached query plans

Table 2. SubqueryExpressions

SubqueryExpression	Description
Exists	
ListQuery	
ScalarSubquery	

`SubqueryExpression` is [resolved](#) when the [children](#) are resolved and the [subquery logical plan](#) is [resolved](#).

references ...FIXME

semanticEquals ...FIXME

canonicalize ...FIXME

hasInOrExistsSubquery Object Method

```
hasInOrExistsSubquery(e: Expression): Boolean
```

hasInOrExistsSubquery ...FIXME

Note	<code>hasInOrExistsSubquery</code> is used when...FIXME
------	---

hasCorrelatedSubquery Object Method

```
hasCorrelatedSubquery(e: Expression): Boolean
```

hasCorrelatedSubquery ...FIXME

Note	<code>hasCorrelatedSubquery</code> is used when...FIXME
------	---

hasSubquery Object Method

```
hasSubquery(e: Expression): Boolean
```

hasSubquery ...FIXME

Note	<code>hasSubquery</code> is used when...FIXME
------	---

Creating SubqueryExpression Instance

`SubqueryExpression` takes the following when created:

- Subquery [logical plan](#)
- Child [expressions](#)
- Expression ID (as `ExprId`)

TimeWindow Unevaluable Unary Expression

`TimeWindow` is an [unevaluable](#) and [non-SQL](#) unary expression that represents window function.

```
import org.apache.spark.sql.functions.window
scala> val timeColumn = window('time, "5 seconds")
timeColumn: org.apache.spark.sql.Column = timewindow(time, 5000000, 5000000, 0) AS `window` 

scala> val timeWindowExpr = timeColumn.expr
timeWindowExpr: org.apache.spark.sql.catalyst.expressions.Expression = timewindow('time
, 5000000, 5000000, 0) AS window#3

scala> println(timeWindowExpr.numberedTreeString)
00 timewindow('time, 5000000, 5000000, 0) AS window#3
01 +- timewindow('time, 5000000, 5000000, 0)
02   +- 'time

import org.apache.spark.sql.catalyst.expressions.TimeWindow
scala> val timeWindow = timeColumn.expr.children.head.asInstanceOf[TimeWindow]
timeWindow: org.apache.spark.sql.catalyst.expressions.TimeWindow = timewindow('time, 5
000000, 5000000, 0)
```

`interval` can include the following units:

- year(s)
- month(s)
- week(s)
- day(s)
- hour(s)
- minute(s)
- second(s)
- millisecond(s)
- microsecond(s)

```
// the most elaborate interval with all the units
interval 0 years 0 months 1 week 0 days 0 hours 1 minute 20 seconds 0 milliseconds 0 microseconds

interval -5 seconds
```

Note The number of months greater than 0 [are not supported](#) for the interval.

`TimeWindow` can never be resolved as it is converted to `Filter` with `Expand` logical operators at [analysis phase](#).

parseExpression Internal Method

```
parseExpression(expr: Expression): Long
```

Caution

FIXME

Analysis Phase

`TimeWindow` is resolved to `Expand` logical operator when [TimeWindowing](#) logical evaluation rule is executed.

```
// https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html
import java.time.LocalDateTime
// https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html
import java.sql.Timestamp
val levels = Seq(
    // (year, month, dayOfMonth, hour, minute, second)
    ((2012, 12, 12, 12, 12, 12), 5),
    ((2012, 12, 12, 12, 12, 14), 9),
    ((2012, 12, 12, 13, 13, 14), 4),
    ((2016, 8, 13, 0, 0, 0), 10),
    ((2017, 5, 27, 0, 0, 0), 15)).
    map { case ((yy, mm, dd, h, m, s), a) => (LocalDateTime.of(yy, mm, dd, h, m, s), a) }
).
    map { case (ts, a) => (Timestamp.valueOf(ts), a) }.
    toDF("time", "level")
scala> levels.show
+-----+----+
|          time|level|
+-----+----+
|2012-12-12 12:12:12|    5|
|2012-12-12 12:12:14|    9|
|2012-12-12 13:13:14|    4|
|2016-08-13 00:00:00|   10|
|2017-05-27 00:00:00|   15|
+-----+----+
val q = levels.select(window($"time", "5 seconds"))

// Before Analyzer
scala> println(q.queryExecution.logical.numberedTreeString)
00 'Project [timewindow('time, 5000000, 5000000, 0) AS window#18]
01 +- Project [_1#6 AS time#9, _2#7 AS level#10]
02     +- LocalRelation [_1#6, _2#7]

// After Analyzer
scala> println(q.queryExecution.analyzed.numberedTreeString)
00 Project [window#19 AS window#18]
01 +- Filter ((time#9 >= window#19.start) && (time#9 < window#19.end))
02     +- Expand [List(named_struct(start, (((CEIL((cast((precisetimestamp(time#9) - 0
) as double) / cast(5000000 as double)) + cast(0 as bigint)) - cast(1 as bigint)) * 5
000000) + 0), end, (((((CEIL((cast((precisetimestamp(time#9) - 0) as double) / cast(50
00000 as double)) + cast(0 as bigint)) - cast(1 as bigint)) * 5000000) + 0) + 5000000
)), time#9, level#10), List(named_struct(start, (((CEIL((cast((precisetimestamp(time#
9) - 0) as double) / cast(5000000 as double)) + cast(1 as bigint)) - cast(1 as bigint
)) * 5000000) + 0), end, (((((CEIL((cast((precisetimestamp(time#9) - 0) as double) / c
ast(5000000 as double)) + cast(1 as bigint)) - cast(1 as bigint)) * 5000000) + 0) + 5
000000)), time#9, level#10)], [window#19, time#9, level#10]
03         +- Project [_1#6 AS time#9, _2#7 AS level#10]
04             +- LocalRelation [_1#6, _2#7]
```

apply Factory Method

```
apply(  
    timeColumn: Expression,  
    windowDuration: String,  
    slideDuration: String,  
    startTime: String): TimeWindow
```

`apply` creates a `TimeWindow` with `timeColumn` `expression` and `windowDuration`, `slideDuration`, `startTime` `microseconds`.

Note

`apply` is used exclusively in `window` function.

Parsing Time Interval to Microseconds — `getIntervalInMicroSeconds` Internal Method

```
getIntervalInMicroSeconds(interval: String): Long
```

`getIntervalInMicroSeconds` parses `interval` string to microseconds.

Internally, `getIntervalInMicroSeconds` adds `interval` prefix to the input `interval` unless it is already available.

`getIntervalInMicroSeconds` creates `CalendarInterval` from the input `interval`.

`getIntervalInMicroSeconds` reports `IllegalArgumentException` when the number of months is greater than `0`.

Note

`getIntervalInMicroSeconds` is used when:

- `TimeWindow` is `created`
- `TimeWindow` does `parseExpression`

TypedAggregateExpression Expression

`TypedAggregateExpression` is the [contract](#) for [AggregateFunction](#) expressions that...FIXME

```
package org.apache.spark.sql.execution.aggregate

trait TypedAggregateExpression extends AggregateFunction {
    // only required methods that have no implementation
    def aggregator: Aggregator[Any, Any, Any]
    def inputClass: Option[Class[_]]
    def inputDeserializer: Option[Expression]
    def inputSchema: Option[StructType]
    def withInputInfo(deser: Expression, cls: Class[_], schema: StructType): TypedAggregateExpression
}
```

`TypedAggregateExpression` is used when:

- `TypedColumn` is requested to [withInputType](#) (for [Dataset.select](#), [KeyValueGroupedDataset.agg](#) and [RelationalGroupedDataset.agg](#) operators)
- `Column` is requested to [generateAlias](#) and [named](#) (for [Dataset.select](#) and [KeyValueGroupedDataset.agg](#) operators)
- `RelationalGroupedDataset` is requested to [alias](#) (when `RelationalGroupedDataset` is requested to [create a DataFrame from aggregate expressions](#))

Table 1. TypedAggregateExpression Contract

Method	Description
<code>aggregator</code>	Aggregator
<code>inputClass</code>	Used when...FIXME
<code>inputDeserializer</code>	Used when...FIXME
<code>inputSchema</code>	Used when...FIXME
<code>withInputInfo</code>	Used when...FIXME

Table 2. TypedAggregateExpressions

Aggregator	Description
ComplexTypedAggregateExpression	
SimpleTypedAggregateExpression	

Creating TypedAggregateExpression — apply Factory Method

```
apply[BUF : Encoder, OUT : Encoder](  
    aggregator: Aggregator[_, BUF, OUT]): TypedAggregateExpression
```

apply ...FIXME

Note

apply is used exclusively when Aggregator is requested to convert itself to a TypedColumn.

TypedImperativeAggregate — Contract for Imperative Aggregate Functions with Custom Aggregation Buffer

`TypedImperativeAggregate` is the [contract](#) for imperative aggregation functions that allows for an arbitrary user-defined java object to be used as [internal aggregation buffer](#).

Table 1. TypedImperativeAggregate as ImperativeAggregate

ImperativeAggregate Method	Description
<code>aggBufferAttributes</code>	
<code>aggBufferSchema</code>	
<code>initialize</code>	Creates an aggregation buffer and puts it at <code>mutableAggBufferOffset</code> position in the input <code>buffer InternalRow</code> .
<code>inputAggBufferAttributes</code>	

Table 2. TypedImperativeAggregate's Direct Implementations

Name	Description
<code>ApproximatePercentile</code>	
<code>Collect</code>	
<code>ComplexTypedAggregateExpression</code>	
<code>CountMinSketchAgg</code>	
<code>HiveUDAFFunction</code>	
<code>Percentile</code>	

TypedImperativeAggregate Contract

```

package org.apache.spark.sql.catalyst.expressions.aggregate

abstract class TypedImperativeAggregate[T] extends ImperativeAggregate {
  def createAggregationBuffer(): T
  def deserialize(storageFormat: Array[Byte]): T
  def eval(buffer: T): Any
  def merge(buffer: T, input: T): T
  def serialize(buffer: T): Array[Byte]
  def update(buffer: T, input: InternalRow): T
}

```

Table 3. TypedImperativeAggregate Contract

Method	Description
createAggregationBuffer	Used exclusively when a <code>TypedImperativeAggregate</code> is initialized
deserialize	
eval	
merge	
serialize	
update	

UnaryExpression Contract

`UnaryExpression` is...FIXME

defineCodeGen Method

```
defineCodeGen(  
    ctx: CodegenContext,  
    ev: ExprCode,  
    f: String => String): ExprCode
```

`defineCodeGen` ...FIXME

Note	<code>defineCodeGen</code> is used when...FIXME
------	---

nullSafeEval Method

```
nullSafeEval(input: Any): Any
```

`nullSafeEval` simply fails with the following error (and is expected to be overrided to save null-check code):

```
UnaryExpressions must override either eval or nullSafeEval
```

Note	<code>nullSafeEval</code> is used exclusively when <code>UnaryExpression</code> is requested to eval .
------	--

Evaluating Expression — eval Method

```
eval(input: InternalRow): Any
```

Note	<code>eval</code> is part of Expression Contract for the interpreted (non-code-generated) expression evaluation , i.e. evaluating a Catalyst expression to a JVM object for a given internal binary row .
------	--

`eval` ...FIXME

UnixTimestamp TimeZoneAware Binary Expression

`UnixTimestamp` is a [binary](#) expression with [timezone](#) support that represents `unix_timestamp` function (and indirectly [to_date](#) and [to_timestamp](#)).

```
import org.apache.spark.sql.functions.unix_timestamp
val c1 = unix_timestamp()

scala> c1.explain(true)
unix_timestamp(current_timestamp(), yyyy-MM-dd HH:mm:ss, None)

scala> println(c1.expr.numberedTreeString)
00 unix_timestamp(current_timestamp(), yyyy-MM-dd HH:mm:ss, None)
01 :- current_timestamp()
02 +- yyyy-MM-dd HH:mm:ss

import org.apache.spark.sql.catalyst.expressions.UnixTimestamp
scala> c1.expr.newInstanceOf[UnixTimestamp]
res0: Boolean = true
```

Note

`UnixTimestamp` is `UnixTime` expression internally (as is `ToUnixTimestamp` expression).

`UnixTimestamp` supports `StringType`, [DateType](#) and `TimestampType` as input types for a time expression and returns `LongType`.

```
scala> c1.expr.eval()
res1: Any = 1493354303
```

`UnixTimestamp` uses `DateTimeUtils.newDateFormat` for date/time format (as Java's [java.text.DateFormat](#)).

UnresolvedAttribute Leaf Expression

`UnresolvedAttribute` is a named [Attribute](#) leaf expression (i.e. it has a name) that represents a reference to an entity in a logical query plan.

`UnresolvedAttribute` is [created](#) when:

- `AstBuilder` is requested to [visitDereference](#)
- `LogicalPlan` is requested to [resolve an attribute by name parts](#)
- `DescribeColumnCommand` is [executed](#)

`UnresolvedAttribute` can never be [resolved](#) (and is replaced at [analysis phase](#)).

Note	<p><code>UnresolvedAttribute</code> is resolved when <code>Analyzer</code> is executed by the following logical resolution rules:</p> <ul style="list-style-type: none"> • ResolveReferences • ResolveMissingReferences • ResolveDeserializer • ResolveSubquery
-------------	---

Given `UnresolvedAttribute` can never be resolved it should not come as a surprise that it [cannot be evaluated](#) either (i.e. produce a value given an internal row). When requested to evaluate, `UnresolvedAttribute` simply reports a `UnsupportedOperationException`.

```
Cannot evaluate expression: [this]
```

`UnresolvedAttribute` takes **name parts** when created.

```
import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute

scala> val t1 = UnresolvedAttribute("t1")
t1: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 't1

scala> val t2 = UnresolvedAttribute("db1.t2")
t2: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 'db1.t2

scala> println(s"Number of name parts: ${t2.nameParts.length}")
Number of name parts: 2

scala> println(s"Name parts: ${t2.nameParts.mkString(",")}")
Name parts: db1,t2
```

`UnresolvedAttribute` can be created with a fully-qualified name with dots to separate name parts.

```
apply(name: String): UnresolvedAttribute
```

Tip

Use backticks (``) around names with dots (.) to disable them as separators.

The following is a two-part attribute name with `a.b` and `c` name parts.

```
`a.b`.c
```

`UnresolvedAttribute` can also be created without the dots with the special meaning.

```
import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
val attr1 = UnresolvedAttribute.quoted("a.b.c")
scala> println(s"Number of name parts: ${attr1.nameParts.length}")
Number of name parts: 1
```

Note

Catalyst DSL defines two Scala implicits to create an `UnresolvedAttribute`:

- `StringToAttributeConversionHelper` is a Scala implicit class that converts `$"colName"` into an `UnresolvedAttribute`
- `symbolToUnresolvedAttribute` is a Scala implicit method that converts `'colName'` into an `UnresolvedAttribute`

Both implicits are part of [ExpressionConversions](#) Scala trait of Catalyst DSL.

Import `expressions` object to get access to the expression conversions.

```
// Use `sbt console` with Spark libraries defined (in `build.sbt`)
import org.apache.spark.sql.catalyst.dsl.expressions._
scala> :type $"name"
org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute

import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
val nameAttr: UnresolvedAttribute = 'name

scala> :type nameAttr
org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
```

Note

A `UnresolvedAttribute` can be replaced by (*resolved*) a `NamedExpression` using an [analyzed logical plan](#) (of the structured query the attribute is part of).

```
val analyzedPlan = Seq((0, "zero")).toDF("id", "name").queryExecution.analyzed

import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
val nameAttr = UnresolvedAttribute("name")
val nameResolved = analyzedPlan.resolveQuoted(
    name = nameAttr.name,
    resolver = spark.sessionState.analyzer.resolver).getOrElse(nameAttr)

scala> println(nameResolved.numberedTreeString)
00 name#47: string

scala> :type nameResolved
org.apache.spark.sql.catalyst.expressions.NamedExpression
```

UnresolvedFunction Unevaluable Expression — Logical Representation of Functions in Queries

`UnresolvedFunction` is an [Catalyst expression](#) that represents a function (application) in a logical query plan.

`UnresolvedFunction` is [created](#) as a result of the following:

- `callUDF` standard function
- `RelationalGroupedDataset.agg` operator with aggregation functions specified by name (that [converts function names to UnresolvedFunction expressions](#))
- `AstBuilder` is requested to [visitFunctionCall](#) (in SQL queries)

`UnresolvedFunction` can never be [resolved](#) (and is replaced at analysis phase).

Note	<code>UnresolvedFunction</code> is first looked up in LookupFunctions logical rule and then resolved in ResolveFunctions logical resolution rule.
------	---

Given `UnresolvedFunction` can never be resolved it should not come as a surprise that it [cannot be evaluated](#) either (i.e. produce a value given an internal row). When requested to evaluate, `UnresolvedFunction` simply reports a `UnsupportedOperationException`.

Cannot evaluate expression: [this]	
------------------------------------	--

Note	Unevaluable expressions are expressions that have to be replaced by some other expressions during analysis or optimization (or they fail analysis).
------	---

Tip	Use Catalyst DSL's <code>function</code> or <code>distinctFunction</code> to create a <code>UnresolvedFunction</code> with <code>isDistinct</code> flag off and on, respectively.
-----	---

```
// Using Catalyst DSL to create UnresolvedFunctions
import org.apache.spark.sql.catalyst.dsl.expressions._

// Scala Symbols supported only
val f = 'f.function()
scala> :type f
org.apache.spark.sql.catalyst.analysis.UnresolvedFunction

scala> f.isDistinct
res0: Boolean = false

val g = 'g.distinctFunction()
scala> g.isDistinct
res1: Boolean = true
```

Creating UnresolvedFunction (With Database Undefined)

— apply Factory Method

```
apply(name: String, children: Seq[Expression], isDistinct: Boolean): UnresolvedFunction
```

`apply` creates a `FunctionIdentifier` with the `name` and no database first and then creates a `UnresolvedFunction` with the `FunctionIdentifier`, `children` and `isDistinct` flag.

Note

`apply` is used when:

- `callUDF` standard function is used
- `RelationalGroupedDataset` is requested to `agg` with aggregation functions specified by name (and converts function names to `UnresolvedFunction expressions`)

Creating UnresolvedFunction Instance

`UnresolvedFunction` takes the following when created:

- `FunctionIdentifier`
- Child `expressions`
- `isDistinct` flag

UnresolvedGenerator Expression

`UnresolvedGenerator` is a [Generator](#) that represents an unresolved generator in a logical query plan.

`UnresolvedGenerator` is [created](#) exclusively when `AstBuilder` is requested to [withGenerate](#) (as part of [Generate](#) logical operator) for SQL's `LATERAL VIEW` (in `SELECT` or `FROM` clauses).

```
// SQLs borrowed from https://cwiki.apache.org/confluence/display/Hive/LanguageManual+
LateralView
val sqlText = """
    SELECT pageid, adid
    FROM pageAds LATERAL VIEW explode(adid_list) adTable AS adid
"""

// Register pageAds table
Seq(
  ("front_page", Array(1, 2, 3)),
  ("contact_page", Array(3, 4, 5)))
  .toDF("pageid", "adid_list")
  .createOrReplaceTempView("pageAds")
val query = sql(sqlText)
val logicalPlan = query.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Project ['pageid, 'adid]
01 +- 'Generate 'explode('adid_list), false, adtable, ['adid]
02   +- 'UnresolvedRelation `pageAds`

import org.apache.spark.sql.catalyst.plans.logical.Generate
val generator = logicalPlan.collectFirst { case g: Generate => g.generator }.get

scala> :type generator
org.apache.spark.sql.catalyst.expressions.Generator

import org.apache.spark.sql.catalyst.analysis.UnresolvedGenerator
scala> generator.isInstanceOf[UnresolvedGenerator]
res1: Boolean = true
```

`UnresolvedGenerator` can never be [resolved](#) (and is replaced at [analysis phase](#)).

Given `UnresolvedGenerator` can never be resolved it should not come as a surprise that it [cannot be evaluated](#) either (i.e. produce a value given an internal row). When requested to evaluate, `UnresolvedGenerator` simply reports a `UnsupportedOperationException`.

```
Cannot evaluate expression: [this]
```

Note

`UnresolvedGenerator` is resolved to a concrete [Generator](#) expression when [ResolveFunctions](#) logical resolution rule is executed.

Note

`UnresolvedGenerator` is similar to [UnresolvedFunction](#) and differs mostly by the type (to make Spark development with Scala easier?)

Creating UnresolvedGenerator Instance

`UnresolvedGenerator` takes the following when created:

- `FunctionIdentifier`
- Child [expressions](#)

UnresolvedOrdinal Unevaluable Leaf Expression

`UnresolvedOrdinal` is a [leaf expression](#) that represents a single integer literal in [Sort](#) logical operators (in [SortOrder](#) ordering expressions) and in [Aggregate](#) logical operators (in [grouping expressions](#)) in a logical plan.

`UnresolvedOrdinal` is [created](#) when `SubstituteUnresolvedOrdinals` logical resolution rule is executed.

```
// Note "order by 1" clause
val sqlText = "select id from VALUES 1, 2, 3 t1(id) order by 1"
val logicalPlan = spark.sql(sqlText).queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Sort [1 ASC NULLS FIRST], true
01 +- 'Project ['id]
02   +- 'SubqueryAlias t1
03     +- 'UnresolvedInlineTable [id], [List(1), List(2), List(3)]

import org.apache.spark.sql.catalyst.analysis.SubstituteUnresolvedOrdinals
val rule = new SubstituteUnresolvedOrdinals(spark.sessionState.conf)

val logicalPlanWithUnresolvedOrdinals = rule.apply(logicalPlan)
scala> println(logicalPlanWithUnresolvedOrdinals.numberedTreeString)
00 'Sort [unresolvedordinal(1) ASC NULLS FIRST], true
01 +- 'Project ['id]
02   +- 'SubqueryAlias t1
03     +- 'UnresolvedInlineTable [id], [List(1), List(2), List(3)]

import org.apache.spark.sql.catalyst.plans.logical.Sort
val sortOp = logicalPlanWithUnresolvedOrdinals.collect { case s: Sort => s }.head
val sortOrder = sortOp.order.head

import org.apache.spark.sql.catalyst.analysis.UnresolvedOrdinal
val unresolvedOrdinalExpr = sortOrder.child.asInstanceOf[UnresolvedOrdinal]
scala> println(unresolvedOrdinalExpr)
unresolvedordinal(1)
```

`UnresolvedOrdinal` takes a single `ordinal` integer when created.

`UnresolvedOrdinal` is an [unevaluable expression](#) and cannot be evaluated (i.e. produce a value given an internal row).

Note

An [unevaluatable expression](#) cannot be evaluated to produce a value (neither in [interpreted](#) nor [code-generated](#) expression evaluations) and has to be resolved (replaced) to some other expressions or logical operators at [analysis](#) or [optimization](#) phases or they fail analysis.

`UnresolvedOrdinal` can never be [resolved](#) (and is replaced at [analysis phase](#)).

Note

`UnresolvedOrdinal` is resolved when [ResolveOrdinalInOrderByAndGroupBy](#) logical resolution rule is executed.

`UnresolvedOrdinal` has [no representation in SQL](#).

Note

`UnresolvedOrdinal` in GROUP BY ordinal position is not allowed for a select list with a star (*).

UnresolvedRegex

UnresolvedRegex is...FIXME

UnresolvedStar Expression

`UnresolvedStar` is a [Star](#) expression that represents a star (i.e. all) expression in a logical query plan.

`UnresolvedStar` is [created](#) when:

- `Column` is [created](#) with `*`
- `AstBuilder` is requested to [visitStar](#)

```
val q = spark.range(5).select(".*")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'Project [*]
01 +- AnalysisBarrier
02     +- Range (0, 5, step=1, splits=Some(8))

import org.apache.spark.sql.catalyst.analysis.UnresolvedStar
val starExpr = plan.expressions.head.asInstanceOf[UnresolvedStar]

val namedExprs = starExpr.expand(input = q.queryExecution.analyzed, spark.sessionState
.analyzer.resolver)
scala> println(namedExprs.head.numberedTreeString)
00 id#0: bigint
```

`UnresolvedStar` can never be [resolved](#), and is [expanded](#) at analysis (when [ResolveReferences](#) logical resolution rule is executed).

Note	<code>UnresolvedStar</code> can only be used in Project , Aggregate or ScriptTransformation logical operators.
------	--

Given `UnresolvedStar` can never be [resolved](#) it should not come as a surprise that it [cannot be evaluated](#) either (i.e. produce a value given an internal row). When requested to evaluate, `UnresolvedStar` simply reports a `UnsupportedOperationException`.

```
Cannot evaluate expression: [this]
```

When created, `UnresolvedStar` takes **name parts** that, once concatenated, is the target of the star expansion.

```
import org.apache.spark.sql.catalyst.analysis.UnresolvedStar
scala> val us = UnresolvedStar(None)
us: org.apache.spark.sql.catalyst.analysis.UnresolvedStar = *

scala> val ab = UnresolvedStar(Some("a" :: "b" :: Nil))
ab: org.apache.spark.sql.catalyst.analysis.UnresolvedStar = List(a, b).*
```

Tip Use `star` operator from Catalyst DSL's [expressions](#) to create an `UnresolvedStar`.

```
import org.apache.spark.sql.catalyst.dsl.expressions._
val s = star()
scala> :type s
org.apache.spark.sql.catalyst.expressions.Expression

import org.apache.spark.sql.catalyst.analysis.UnresolvedStar
assert(s.isInstanceOf[UnresolvedStar])

val s = star("a", "b")
scala> println(s)
WrappedArray(a, b).*
```

You could also use `""` or `'` to create an `UnresolvedStar`, but that requires `sbt console` (with Spark libraries defined in `build.sbt`) as the Catalyst DSL `expressions` implicits interfere with the Spark implicits to create columns.

Note

AstBuilder [replaces](#) `count(*)` (with no `DISTINCT` keyword) to `count(1)`.

```
val q = sql("SELECT COUNT(*) FROM RANGE(1,2,3)")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'Project [unresolvedalias('count(1), None)]
01 +- 'UnresolvedTableValuedFunction range, [1, 2, 3]

val q = sql("SELECT COUNT(DISTINCT *) FROM RANGE(1,2,3)")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'Project [unresolvedalias('COUNT(*), None)]
01 +- 'UnresolvedTableValuedFunction RANGE, [1, 2, 3]
```

Star Expansion — `expand` Method

```
expand(input: LogicalPlan, resolver: Resolver): Seq[NamedExpression]
```

Note

`expand` is part of [Star Contract](#) to...FIXME.

`expand` first expands to named expressions per [target](#):

- For unspecified [target](#), `expand` gives the [output](#) schema of the `input` logical query plan (that assumes that the star refers to a relation / table)

- For `target` with one element, `expand` gives the table (attribute) in the `output` schema of the `input` logical query plan (using `qualifiers`) if available

With no result earlier, `expand` then requests the `input` logical query plan to `resolve` the `target` name parts to a named expression.

For a named expression of `StructType` data type, `expand` creates an `Alias` expression with a `GetStructField` unary expression (with the resolved named expression and the field index).

```
val q = Seq((0, "zero")).toDF("id", "name").select(struct("id", "name") as "s")
val analyzedPlan = q.queryExecution.analyzed

import org.apache.spark.sql.catalyst.analysis.UnresolvedStar
import org.apache.spark.sql.catalyst.dsl.expressions._
val s = star("s").asInstanceOf[UnresolvedStar]
val exprs = s.expand(input = analyzedPlan, spark.sessionState.analyzer.resolver)

// star("s") should expand to two Alias(GetStructField) expressions
// s is a struct of id and name in the query

import org.apache.spark.sql.catalyst.expressions.{Alias, GetStructField}
val getStructFields = exprs.collect { case Alias(g: GetStructField, _) => g }.map(_.sq
1)
scala> getStructFields.foreach(println)
`s`.`id`
`s`.`name`
```

`expand` reports a `AnalysisException` when:

- The `data type` of the named expression (when the `input` logical plan was requested to `resolve` the `target`) is not a `StructType`.

```
Can only star expand struct data types. Attribute: `'[target]'`
```

- Earlier attempts gave no results

```
cannot resolve '[target].*' given input columns '[from]'
```

UnresolvedWindowExpression Unevaluable Expression — WindowExpression With Unresolved Window Specification Reference

`UnresolvedWindowExpression` is an [unevaluable expression](#) that represents...FIXME

Note

An [unevaluable expression](#) cannot be evaluated to produce a value (neither in [interpreted](#) nor [code-generated](#) expression evaluations) and has to be resolved (replaced) to some other expressions or logical operators at [analysis](#) or [optimization](#) phases or they fail analysis.

`UnresolvedWindowExpression` is [created](#) when:

- FIXME

`UnresolvedWindowExpression` is created to represent a `child expression` and `WindowSpecReference` (with an identifier for the window reference) when `AstBuilder` parses a function evaluated in a windowed context with a `WindowSpecReference`.

`UnresolvedWindowExpression` is resolved to a `WindowExpression` when `Analyzer` resolves `UnresolvedWindowExpressions`.

```
import spark.sessionState.sqlParser

scala> sqlParser.parseExpression("foo() OVER windowSpecRef")
res1: org.apache.spark.sql.catalyst.expressions.Expression = unresolvedwindowexpression('foo(), WindowSpecReference(windowSpecRef))
```

Table 1. `UnresolvedWindowExpression`'s Properties

Name	Description
<code>dataType</code>	Reports a <code>unresolvedException</code>
<code>foldable</code>	Reports a <code>unresolvedException</code>
<code>nullable</code>	Reports a <code>unresolvedException</code>
<code>resolved</code>	Disabled (i.e. <code>false</code>)

WindowExpression Unevaluable Expression

`WindowExpression` is an [unevaluable expression](#) that represents a [window function](#) (over some [WindowSpecDefinition](#)).

Note

An [unevaluable expression](#) cannot be evaluated to produce a value (neither in [interpreted](#) nor [code-generated](#) expression evaluations) and has to be resolved (replaced) to some other expressions or logical operators at [analysis](#) or [optimization](#) phases or they fail analysis.

`WindowExpression` is created when:

- `WindowSpec` is requested to [withAggregate](#) (when `Column.over` operator is used)
- `WindowsSubstitution` logical evaluation rule is [executed](#) (with `WithWindowDefinition` logical operators with `UnresolvedWindowExpression` expressions)
- `AstBuilder` is requested to [parse a function call](#) in a SQL statement

Note

`WindowExpression` can only be [created](#) with `AggregateExpression`, `AggregateWindowFunction` or `OffsetWindowFunction` expressions which is enforced at [analysis](#).

```
// Using Catalyst DSL
val wf = 'count.function(star())
val windowSpec = ???
```

Note

`WindowExpression` is resolved in [ExtractWindowExpressions](#), [ResolveWindowFrame](#) and [ResolveWindowOrder](#) logical rules.

```

import org.apache.spark.sql.catalyst.expressions.WindowExpression
// relation - Dataset as a table to query
val table = spark.emptyDataset[Int]

scala> val windowExpr = table
    .selectExpr("count() OVER (PARTITION BY value) AS count")
    .queryExecution
    .logical      (1)
    .expressions
    .toList(0)
    .children(0)
    .asInstanceOf[WindowExpression]
windowExpr: org.apache.spark.sql.catalyst.expressions.WindowExpression = 'count() wind
owspecdefinition('value, UnspecifiedFrame)

scala> windowExpr.sql
res2: String = count() OVER (PARTITION BY `value` UnspecifiedFrame)

```

1. Use `sqlParser` directly as in [WithWindowDefinition Example](#)

Table 1. WindowExpression's Properties

Name	Description
<code>children</code>	Collection of two expressions , i.e. windowFunction and WindowSpecDefinition , for which <code>WindowExpression</code> was created.
<code>dataType</code>	DataType of windowFunction
<code>foldable</code>	Whether or not windowFunction is foldable.
<code>nullable</code>	Whether or not windowFunction is nullable.
<code>sql</code>	<code>"[windowFunction].sql OVER [windowSpec].sql"</code>
<code>toString</code>	<code>"[windowFunction] [windowSpec]"</code>
Note	<code>WindowExpression</code> is subject to NullPropagation and DecimalAggregates logical optimizations.
Note	Distinct window functions are not supported which is enforced at analysis .
Note	An offset window function can only be evaluated in an ordered row-based window frame with a single offset which is enforced at analysis .

Catalyst DSL — `windowExpr` Operator

```
windowExpr(windowFunc: Expression, windowSpec: WindowSpecDefinition): WindowExpression
```

`windowExpr` operator in Catalyst DSL creates a [WindowExpression](#) expression, e.g. for testing or Spark SQL internals exploration.

```
// FIXME: DEMO
```

Creating WindowExpression Instance

`WindowExpression` takes the following when created:

- Window function [expression](#)
- [WindowSpecDefinition](#) expression

WindowFunction Contract — Window Function Expressions With WindowFrame

`WindowFunction` is the [contract](#) of [function expressions](#) that define a [WindowFrame](#) in which the window operator must be executed.

```
package org.apache.spark.sql.catalyst.expressions

trait WindowFunction extends Expression {
    // No required properties (vals and methods) that have no implementation
}
```

Table 1. WindowFunctions (Direct Implementations)

WindowFunction	Description
AggregateWindowFunction	
OffsetWindowFunction	

Defining WindowFrame for Execution — `frame` Method

```
frame: WindowFrame
```

`frame` defines the `WindowFrame` for function execution, i.e. the `WindowFrame` in which the window operator must be executed.

`frame` is `UnspecifiedFrame` by default.

Note	<code>frame</code> is used when...FIXME
------	---

WindowSpecDefinition Unevaluable Expression

`WindowSpecDefinition` is an [unevaluable expression](#) (i.e. with no support for `eval` and `doGenCode` methods).

`WindowSpecDefinition` is [created](#) when:

- `AstBuilder` is requested to [parse a window specification](#) in a SQL query
- `Column.over` operator is used

```
import org.apache.spark.sql.expressions.Window
val byValueDesc = Window.partitionBy("value").orderBy($"value".desc)

val q = table.withColumn("count over window", count("*") over byValueDesc)

import org.apache.spark.sql.catalyst.expressions.WindowExpression
val windowExpr = q.queryExecution
  .logical
  .expressions(1)
  .children(0)
  .asInstanceOf[WindowExpression]

scala> windowExpr.windowSpec
res0: org.apache.spark.sql.catalyst.expressions.WindowSpecDefinition = windowspecdefinition('value, 'value DESC NULLS LAST, UnspecifiedFrame)
```

```
import org.apache.spark.sql.catalyst.expressions.WindowSpecDefinition

Seq((0, "hello"), (1, "windows"))
  .toDF("id", "token")
  .createOrReplaceTempView("mytable")

val sqlText = """
  SELECT count(*) OVER myWindowSpec
  FROM mytable
  WINDOW
    myWindowSpec AS (
      PARTITION BY token
      ORDER BY id
      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    )
"""

import spark.sessionState.{analyzer, sqlParser}
```

```

scala> val parsedPlan = sqlParser.parsePlan(sqlText)
parsedPlan: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'WithWindowDefinition Map(myWindowSpec -> windowspecdefinition('token, 'id ASC NULLS F
IRST, RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW))
+- 'Project [unresolvedalias(unresolvedwindowexpression('count(1), WindowSpecReference
(myWindowSpec)), None)]
  +- 'UnresolvedRelation `mytable`

import org.apache.spark.sql.catalyst.plans.logical.WithWindowDefinition
val myWindowSpec = parsedPlan.asInstanceOf[WithWindowDefinition].windowDefinitions("my
WindowSpec")

scala> println(myWindowSpec)
windowspecdefinition('token, 'id ASC NULLS FIRST, RANGE BETWEEN UNBOUNDED PRECEDING AN
D CURRENT ROW)

scala> println(myWindowSpec.sql)
(PARTITION BY `token` ORDER BY `id` ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW)

scala> sql(sqlText)
res4: org.apache.spark.sql.DataFrame = [count(1) OVER (PARTITION BY token ORDER BY id
ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW): bigint]

scala> println(analyzer.execute(sqlParser.parsePlan(sqlText)))
Project [count(1) OVER (PARTITION BY token ORDER BY id ASC NULLS FIRST RANGE BETWEEN U
NBOUNDED PRECEDING AND CURRENT ROW)#25L]
+- Project [token#13, id#12, count(1) OVER (PARTITION BY token ORDER BY id ASC NULLS F
IRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)#25L, count(1) OVER (PARTITION
BY token ORDER BY id ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)#25L]
  +- Window [count(1) windowspecdefinition(token#13, id#12 ASC NULLS FIRST, RANGE BET
WEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS count(1) OVER (PARTITION BY token ORDER B
Y id ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)#25L], [token#1
3], [id#12 ASC NULLS FIRST]
    +- Project [token#13, id#12]
      +- SubqueryAlias mytable
        +- Project [_1#9 AS id#12, _2#10 AS token#13]
          +- LocalRelation [_1#9, _2#10]

```

Table 1. WindowSpecDefinition's Properties

Name	Description
children	Window <code>partition</code> and <code>order</code> specifications (for which <code>WindowExpression</code> was created).
dataType	Unsupported (i.e. reports a <code>UnsupportedOperationException</code>)
foldable	Disabled (i.e. <code>false</code>)
nullable	Enabled (i.e. <code>true</code>)
resolved	Enabled when <code>children</code> are valid and the input <code>DataType</code> is a <code>SpecifiedWindowFrame</code> .
sql	Contains <code>PARTITION BY</code> with comma-separated elements of <code>partitionSpec</code> (if defined) with <code>ORDER BY</code> with comma-separated elements of <code>orderSpec</code> (if defined) followed by <code>frameSpecification</code> . <code>(PARTITION BY `token` ORDER BY `id` ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)</code>

Creating WindowSpecDefinition Instance

`WindowSpecDefinition` takes the following when created:

- `Expressions` for window partition specification
- Window order specifications (as `SortOrder` unary expressions)
- Window frame specification (as `WindowFrame`)

Validating Data Type Of Window Order-- isValidFrameType Internal Method

```
isValidFrameType(ft: DataType): Boolean
```

`isValidFrameType` is positive (`true`) when the data type of the `window order specification` and the input `ft` `data type` are as follows:

- [DateType](#) and [IntegerType](#)
- [TimestampType](#) and [CalendarIntervalType](#)
- Equal

Otherwise, `isValidFrameType` is negative (`false`).

Note

`isValidFrameType` is used exclusively when `WindowSpecDefinition` is requested to [checkInputDataTypes](#) (with `RangeFrame` as the [window frame specification](#))

Checking Input Data Types — `checkInputDataTypes` Method

`checkInputDataTypes(): TypeCheckResult`

Note

`checkInputDataTypes` is part of the [Expression Contract](#) to checks the input data types.

`checkInputDataTypes` ...FIXME

LogicalPlan Contract — Logical Relational Operator with Children and Expressions / Logical Query Plan

`LogicalPlan` is an extension of the [QueryPlan contract](#) for [logical operators](#) to build a [logical query plan](#) (i.e. a tree of logical operators).

Note

A logical query plan is a tree of [nodes](#) of logical operators that in turn can have (trees of) [Catalyst expressions](#). In other words, there are *at least* two trees at every level (operator).

`LogicalPlan` can be [resolved](#).

In order to get the [logical plan](#) of a structured query you should use the [QueryExecution](#).

```
scala> :type q
org.apache.spark.sql.Dataset[Long]

val plan = q.queryExecution.logical
scala> :type plan
org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
```

`LogicalPlan` goes through [execution stages](#) (as a [QueryExecution](#)). In order to convert a `LogicalPlan` to a `QueryExecution` you should use `SessionState` and request it to "execute" the plan.

```
scala> :type spark
org.apache.spark.sql.SparkSession

// You could use Catalyst DSL to create a logical query plan
scala> :type plan
org.apache.spark.sql.catalyst.plans.logical.LogicalPlan

valqe = spark.sessionState.executePlan(plan)
scala> :typeqe
org.apache.spark.sql.execution.QueryExecution
```

Note

A common idiom in Spark SQL to make sure that a logical plan can be analyzed is to request a `SparkSession` for the `SessionState` that is in turn requested to `execute` the logical plan (which simply creates a `QueryExecution`).

```
scala> :type plan
org.apache.spark.sql.catalyst.plans.logical.LogicalPlan

val qe = sparkSession.sessionState.executePlan(plan)
qe.assertAnalyzed()
// the following gives the analyzed logical plan
// no exceptions are expected since analysis went fine
val analyzedPlan = qe.analyzed
```

Note

Another common idiom in Spark SQL to convert a `LogicalPlan` into a `Dataset` is to use `Dataset.ofRows` internal method that `executes` the logical plan followed by creating a `Dataset` with the `QueryExecution` and a `RowEncoder`.

A logical operator is considered **partially resolved** when its `child operators` are resolved (aka *children resolved*).

A logical operator is (fully) **resolved** to a specific schema when all `expressions` and the `children` are resolved.

```
scala> plan.resolved
res2: Boolean = true
```

A logical plan knows the size of objects that are results of query operators, like `join`, through `statistics` object.

```
scala> val stats = plan.statistics
stats: org.apache.spark.sql.catalyst.plans.logical.Statistics = Statistics(8, false)
```

A logical plan knows the maximum number of records it can compute.

```
scala> val maxRows = plan.maxRows
maxRows: Option[Long] = None
```

`LogicalPlan` can be `streaming` if it contains one or more `structured streaming sources`.

Note

`LogicalPlan` is in the end transformed to a `physical query plan`.

Table 1. Logical Operators / Specialized Logical Plans

LogicalPlan	Description
LeafNode	Logical operator with no <code>child</code> operators
UnaryNode	Logical plan with a single <code>child</code> logical operator
BinaryNode	Logical operator with two <code>child</code> logical operators
Command	
RunnableCommand	

Table 2. LogicalPlan's Internal Registries and Counters

Name	Description
<code>statsCache</code>	Cached plan statistics (as <code>statistics</code>) of the <code>LogicalPlan</code> Computed and cached in <code>stats</code> . Used in <code>stats</code> and <code>verboseStringWithSuffix</code> . Reset in <code>invalidateStatsCache</code>

Getting Cached or Calculating Estimated Statistics — `stats` Method

```
stats(conf: CatalystConf): Statistics
```

`stats` returns the `cached plan statistics` or `computes a new one` (and caches it as `statsCache`).

Note

- `stats` is used when:
- A `LogicalPlan` computes statistics
 - `QueryExecution` builds complete text representation
 - `JoinSelection` checks whether a plan can be broadcast et al
 - `CostBasedJoinReorder` attempts to reorder inner joins
 - `LimitPushDown` is executed (for `FullOuter` join)
 - `AggregateEstimation` estimates Statistics
 - `FilterEstimation` estimates child Statistics
 - `InnerOuterEstimation` estimates Statistics of the left and right sides of a join
 - `LeftSemiAntiEstimation` estimates Statistics
 - `ProjectEstimation` estimates Statistics

`invalidateStatsCache` method

Caution

FIXME

`verboseStringWithSuffix` method

Caution

FIXME

`setAnalyzed` method

Caution

FIXME

Is Logical Plan Streaming? — `isStreaming` method

`isStreaming: Boolean`

`isStreaming` is part of the public API of `LogicalPlan` and is enabled (i.e. `true`) when a logical plan is a [streaming source](#).

By default, it walks over subtrees and calls itself, i.e. `isStreaming`, on every child node to find a streaming source.

```
val spark: SparkSession = ...

// Regular dataset
scala> val ints = spark.createDataset(0 to 9)
ints: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> ints.queryExecution.logical.isStreaming
res1: Boolean = false

// Streaming dataset
scala> val logs = spark.readStream.format("text").load("logs/*.out")
logs: org.apache.spark.sql.DataFrame = [value: string]

scala> logs.queryExecution.logical.isStreaming
res2: Boolean = true
```

Note

Streaming Datasets are part of Structured Streaming.

Refreshing Child Logical Plans — `refresh` Method

```
refresh(): Unit
```

`refresh` calls itself recursively for every `child` logical operator.

Note

`refresh` is overridden by `LogicalRelation` only (that refreshes the location of `HadoopFsRelation` relations only).

Note

`refresh` is used when:

- `SessionCatalog` is requested to `refresh` a table
- `CatalogImpl` is requested to `refresh` a table

resolveQuoted Method

```
resolveQuoted(
  name: String,
  resolver: Resolver): Option[NamedExpression]
```

`resolveQuoted` ...FIXME

Note

`resolveQuoted` is used when...FIXME

Resolving Attribute By Name Parts — `resolve` Method

```
resolve(schema: StructType, resolver: Resolver): Seq[Attribute]
resolve(
  nameParts: Seq[String],
  resolver: Resolver): Option[NamedExpression]
resolve(
  nameParts: Seq[String],
  input: Seq[Attribute],
  resolver: Resolver): Option[NamedExpression] (1)
```

1. A protected method

`resolve ...FIXME`

Note

`resolve` is used when...FIXME

Command Contract — Eagerly-Executed Logical Operator

`Command` is the **marker interface** for logical operators that represent non-query commands that are executed early in the [query plan lifecycle](#) (unlike logical plans in general).

Note	<code>Command</code> is executed when a <code>Dataset</code> is requested for the logical plan (which is after the query has been analyzed).
------	---

`Command` has no [output schema](#) by default.

`Command` has no child logical operators (which makes it similar to [leaf logical operators](#)).

Table 1. Commands (Direct Implementations)

Command	Description
DataWritingCommand	
RunnableCommand	

RunnableCommand Contract — Generic Logical Command with Side Effects

`RunnableCommand` is the generic [logical command](#) that is [executed](#) eagerly for its side effects.

`RunnableCommand` defines one abstract method `run` that computes a collection of [Row](#) records with the side effect, i.e. the result of executing a command.

```
run(sparkSession: SparkSession): Seq[Row]
```

Note	<code>RunnableCommand</code> logical operator is resolved to ExecutedCommandExec physical operator in BasicOperators execution planning strategy.
------	---

Note	<p><code>run</code> is executed when:</p> <ul style="list-style-type: none"> • <code>ExecutedCommandExec</code> executes logical <code>RunnableCommand</code> and caches the result as InternalRows • <code>InsertIntoHadoopFsRelationCommand</code> is executed • <code>QueryExecution</code> is requested to transform the result of executing DescribeTableCommand to a Hive-compatible output format
------	---

Table 1. Available RunnableCommands

RunnableCommand	Description
AddFileCommand	
AddJarCommand	
AlterDatabasePropertiesCommand	
AlterTableAddPartitionCommand	
AlterTableChangeColumnCommand	
AlterTableDropPartitionCommand	
AlterTableRecoverPartitionsCommand	
AlterTableRenameCommand	
AlterTableRenamePartitionCommand	

AlterTableSerDePropertiesCommand					
AlterTableSetLocationCommand					
AlterTableSetPropertiesCommand					
AlterTableUnsetPropertiesCommand					
AlterViewAsCommand					
AnalyzeColumnCommand					
AnalyzePartitionCommand					
AnalyzeTableCommand					
CacheTableCommand	<p>When executed, CacheTableCommand creates registering a temporary view for the optional query.</p> <pre>CACHE [LAZY? TABLE] [table] [AS? [query]]?</pre> <p>CacheTableCommand requests the session-specific catalog for the table.</p> <table border="1"> <tr> <td>Note</td> <td>CacheTableCommand uses SparkSession's Catalog.</td> </tr> </table> <p>If the caching is not LAZY (which is not by default), CacheTableCommand creates a DataFrame from the rows (that will trigger the caching).</p> <table border="1"> <tr> <td>Note</td> <td>CacheTableCommand uses a Spark DataFrame caching by executing the query.</td> </tr> </table> <pre>val q = "CACHE TABLE ids AS SELECT * FROM `ids`" scala> println(sql(q).queryExecution.log) 00 CacheTableCommand `ids`, false 01 +- 'Project [*]' 02 +- 'UnresolvedTableValuedFunction' 'ids' val q2 = "CACHE LAZY TABLE ids" scala> println(sql(q2).queryExecution.log) 17/05/17 06:16:39 WARN CacheManager: Asked for data. 00 CacheTableCommand `ids`, true</pre>	Note	CacheTableCommand uses SparkSession's Catalog.	Note	CacheTableCommand uses a Spark DataFrame caching by executing the query.
Note	CacheTableCommand uses SparkSession's Catalog.				
Note	CacheTableCommand uses a Spark DataFrame caching by executing the query.				
ClearCacheCommand					
CreateDatabaseCommand					

CreateDataSourceTableAsSelectCommand	When executed , ...FIXME Used exclusively when DataSourceAnalysis rule resolves a CreateTable logical operator. Hive table providers (which is when DataFrame to a non-Hive table or for CreateTableAsSelect statements)
CreateDataSourceTableCommand	
CreateFunctionCommand	
CreateHiveTableAsSelectCommand	
CreateTableCommand	
CreateTableLikeCommand	
CreateTempViewUsing	
CreateViewCommand	
DescribeColumnCommand	
DescribeDatabaseCommand	
DescribeFunctionCommand	
DescribeTableCommand	
DropDatabaseCommand	
DropFunctionCommand	
DropTableCommand	
ExplainCommand	
InsertIntoDataSourceCommand	
InsertIntoHadoopFsRelationCommand	
InsertIntoHiveTable	
ListFilesCommand	

ListJarsCommand	
LoadDataCommand	
RefreshResource	
RefreshTable	
ResetCommand	
SaveIntoDataSourceCommand	<p>When executed, requests <code>DataSource</code> to write to <code>source</code> per <code>saveMode</code>.</p> <p>Used exclusively when <code>DataStreamWriter</code> is DataFrame to a data source.</p>
SetCommand	
SetDatabaseCommand	
ShowColumnsCommand	
ShowCreateTableCommand	
ShowDatabasesCommand	
ShowFunctionsCommand	
ShowPartitionsCommand	
ShowTablePropertiesCommand	
ShowTablesCommand	
StreamingExplainCommand	
TruncateTableCommand	
UncacheTableCommand	

DataWritingCommand Contract — Logical Commands That Write Query Data

`DataWritingCommand` is an [extension](#) of the [Command contract](#) for [logical commands](#) that write the result of executing [query](#) (*query data*) to a relation when [executed](#).

`DataWritingCommand` is resolved to a [DataWritingCommandExec](#) physical operator when [BasicOperators](#) execution planning strategy is executed (i.e. plan a [logical plan](#) to a [physical plan](#)).

Table 1. DataWritingCommand Contract

Property	Description
<code>outputColumnNames</code>	<pre>outputColumnNames: Seq[String]</pre> <p>The output column names of the analyzed input query plan Used when <code>DataWritingCommand</code> is requested for the outputColumns</p>
<code>query</code>	<pre>query: LogicalPlan</pre> <p>The analyzed logical query plan representing the data to write (i.e. whose result will be inserted into a relation) Used when <code>DataWritingCommand</code> is requested for the child nodes and outputColumns.</p>
<code>run</code>	<pre>run(sparkSession: SparkSession, child: SparkPlan): Seq[Row]</pre> <p>Executes the command to write query data Used when:</p> <ul style="list-style-type: none"> • <code>DataWritingCommandExec</code> physical operator is requested for the sideEffectResult • <code>DataSource</code> is requested to write data to a data source per save mode followed by reading rows back (when CreateDataSourceTableAsSelectCommand logical command is executed)

When requested for the [child nodes](#), `DataWritingCommand` simply returns the [logical query plan](#).

`DataWritingCommand` defines custom performance metrics.

Table 2. DataWritingCommand's Performance Metrics

Key	Name (in web UI)	Description
<code>numFiles</code>	number of written files	
<code>numOutputBytes</code>	bytes of written output	
<code>numOutputRows</code>	number of output rows	
<code>numParts</code>	number of dynamic part	

The performance metrics are used when:

- `DataWritingCommand` is requested for the `BasicWriteJobStatsTracker`
- `DataWritingCommandExec` physical operator is requested for the metrics

Table 3. DataWritingCommands (Direct Implementations)

DataWritingCommand	Description
<code>CreateDataSourceTableAsSelectCommand</code>	
<code>CreateHiveTableAsSelectCommand</code>	
<code>InsertIntoHadoopFsRelationCommand</code>	
<code>SaveAsHiveFile</code>	Contract for commands that write query result as Hive files (e.g. <code>InsertIntoHiveDirCommand</code> , <code>InsertIntoHiveTable</code>)

basicWriteJobStatsTracker Method

```
basicWriteJobStatsTracker(hadoopConf: Configuration): BasicWriteJobStatsTracker
```

`basicWriteJobStatsTracker` simply creates and returns a new `BasicWriteJobStatsTracker` (with the given Hadoop Configuration and the metrics).

Note	<p><code>basicWriteJobStatsTracker</code> is used when:</p> <ul style="list-style-type: none">• <code>SaveAsHiveFile</code> is requested to <code>saveAsHiveFile</code> (when <code>InsertIntoHiveDirCommand</code> and <code>InsertIntoHiveTable</code> logical commands are executed)• <code>InsertIntoHadoopFsRelationCommand</code> logical command is executed
------	--

Output Columns — `outputColumns` Method

```
outputColumns: Seq[Attribute]
```

`outputColumns` ...FIXME

Note	<p><code>outputColumns</code> is used when:</p> <ul style="list-style-type: none">• <code>CreateHiveTableAsSelectCommand</code>, <code>InsertIntoHiveDirCommand</code> and <code>InsertIntoHadoopFsRelationCommand</code> logical commands are executed• <code>SaveAsHiveFile</code> is requested to <code>saveAsHiveFile</code>
------	---

SaveAsHiveFile Contract — DataWritingCommands That Write Query Result As Hive Files

`SaveAsHiveFile` is the extension of the `DataWritingCommand` contract for [commands](#) that [saveAsHiveFile](#).

Table 1. SaveAsHiveFiles

SaveAsHiveFile	Description
InsertIntoHiveDirCommand	
InsertIntoHiveTable	

saveAsHiveFile Method

```
saveAsHiveFile(
    sparkSession: SparkSession,
    plan: SparkPlan,
    hadoopConf: Configuration,
    fileSinkConf: FileSinkDesc,
    outputLocation: String,
    customPartitionLocations: Map[TablePartitionSpec, String] = Map.empty,
    partitionAttributes: Seq[Attribute] = Nil): Set[String]
```

`saveAsHiveFile ...FIXME`

	<code>saveAsHiveFile</code> is used when: <ul style="list-style-type: none"> • InsertIntoHiveDirCommand logical command is executed • InsertIntoHiveTable logical command is executed (and does processInsert)
Note	

Aggregate Unary Logical Operator

Aggregate is a [unary logical operator](#) that holds the following:

- Grouping [expressions](#)
- Aggregate [named expressions](#)
- Child [logical plan](#)

Aggregate is created to represent the following (after a logical plan is [analyzed](#)):

- SQL's [GROUP BY](#) clause (possibly with `WITH CUBE` or `WITH ROLLUP`)
- [RelationalGroupedDataset](#) aggregations (e.g. [pivot](#))
- [KeyValueGroupedDataset](#) aggregations
- [AnalyzeColumnCommand](#) logical command

Note

Aggregate logical operator is translated to one of [HashAggregateExec](#), [ObjectHashAggregateExec](#) or [SortAggregateExec](#) physical operators in [Aggregation](#) execution planning strategy.

Table 1. Aggregate's Properties

Name	Description			
maxRows	<p>Child logical plan's <code>maxRows</code></p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>Note</td> <td>Part of LogicalPlan contract.</td> </tr> </table>		Note	Part of LogicalPlan contract .
Note	Part of LogicalPlan contract .			
output	<p>Attributes of aggregate named expressions</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>Note</td> <td>Part of QueryPlan contract.</td> </tr> </table>		Note	Part of QueryPlan contract .
Note	Part of QueryPlan contract .			
resolved	<p>Enabled when:</p> <ul style="list-style-type: none"> • expressions and child logical plan are resolved • No WindowExpressions exist in aggregate named expressions <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>Note</td> <td>Part of LogicalPlan contract.</td> </tr> </table>		Note	Part of LogicalPlan contract .
Note	Part of LogicalPlan contract .			
validConstraints	<p>The (expression) constraints of child logical plan and non-aggregate aggregate named expressions.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>Note</td> <td>Part of QueryPlan contract.</td> </tr> </table>		Note	Part of QueryPlan contract .
Note	Part of QueryPlan contract .			

Rule-Based Logical Query Optimization Phase

[PushDownPredicate](#) logical plan optimization applies so-called **filter pushdown** to a [Pivot](#) operator when under `Filter` operator and with all expressions deterministic.

```

import org.apache.spark.sql.catalyst.optimizer.PushDownPredicate

val q = visits
  .groupBy("city")
  .pivot("year")
  .count()
  .where($"city" === "Boston")

val pivotPlanAnalyzed = q.queryExecution.analyzed
scala> println(pivotPlanAnalyzed.numberedTreeString)
00 Filter (city#8 = Boston)
01 +- Project [city#8, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[0] AS
  2015#143L, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[1] AS 2016#144L
, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[2] AS 2017#145L]
02   +- Aggregate [city#8], [city#8, pivotfirst(year#9, count(1) AS `count`#134L, 2015
, 2016, 2017, 0, 0) AS __pivot_count(1) AS `count` AS `count(1) AS ``count````#142]
03     +- Aggregate [city#8, year#9], [city#8, year#9, count(1) AS count(1) AS `coun
t`#134L]
04       +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
05         +- LocalRelation [_1#3, _2#4, _3#5]

val afterPushDown = PushDownPredicate(pivotPlanAnalyzed)
scala> println(afterPushDown.numberedTreeString)
00 Project [city#8, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[0] AS 2
015#143L, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[1] AS 2016#144L,
__pivot_count(1) AS `count` AS `count(1) AS ``count````#142[2] AS 2017#145L]
01 +- Aggregate [city#8], [city#8, pivotfirst(year#9, count(1) AS `count`#134L, 2015,
2016, 2017, 0, 0) AS __pivot_count(1) AS `count` AS `count(1) AS ``count````#142]
02   +- Aggregate [city#8, year#9], [city#8, year#9, count(1) AS count(1) AS `count`#
134L]
03     +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
04       +- Filter (_2#4 = Boston)
05         +- LocalRelation [_1#3, _2#4, _3#5]

```

AlterViewAsCommand Logical Command

`AlterViewAsCommand` is a [logical command](#) for `ALTER VIEW` SQL statement to alter a view.

`AlterViewAsCommand` works with a table identifier (as `TableIdentifier`), the original SQL text, and a [LogicalPlan](#) for the SQL query.

Note	<code>AlterViewAsCommand</code> is described by <code>alterViewQuery</code> labeled alternative in statement expression in <code>SqlBase.g4</code> and parsed using SparkSqlParser .
------	--

When [executed](#), `AlterViewAsCommand` attempts to [alter a temporary view](#) in the current [SessionCatalog](#) first, and if that "fails", [alters the permanent view](#).

Executing Logical Command — `run` Method

```
run(session: SparkSession): Seq[Row]
```

Note	<code>run</code> is part of RunnableCommand Contract to execute (run) a logical command.
------	--

`run ...FIXME`

`alterPermanentView` Internal Method

```
alterPermanentView(session: SparkSession, analyzedPlan: LogicalPlan): Unit
```

`alterPermanentView ...FIXME`

Note	<code>alterPermanentView</code> is used when...FIXME
------	--

AnalysisBarrier Leaf Logical Operator — Hiding Child Query Plan in Analysis

`AnalysisBarrier` is a [leaf logical operator](#) that is a wrapper of an [analyzed logical plan](#) to hide it from the Spark Analyzer. The purpose of `AnalysisBarrier` is to prevent the child logical plan from being analyzed again (and increasing the time spent on query analysis).

`AnalysisBarrier` is [created](#) when:

- `ResolveReferences` logical resolution rule is requested to `dedupRight`
- `ResolveMissingReferences` logical resolution rule is requested to `resolveExprsAndAddMissingAttrs`
- `Dataset` is [created](#)
- `DataFrameWriter` is requested to [execute a logical command](#) for writing to a data source `V1` (when `DataFrameWriter` is requested to [save](#) the rows of a structured query (a `DataFrame`) to a data source)
- `KeyValueGroupedDataset` is requested for the [logical query plan](#)

`AnalysisBarrier` takes a single `child` [logical query plan](#) when created.

`AnalysisBarrier` returns the [child logical query plan](#) when requested for the [inner nodes](#) (that should be shown as an inner nested tree of this node).

`AnalysisBarrier` simply requests the [child logical query plan](#) for the [output schema attributes](#).

`AnalysisBarrier` simply requests the [child logical query plan](#) for the [isStreaming](#) flag.

`AnalysisBarrier` simply requests the [child logical operator](#) for the [canonicalized version](#).

AnalyzeColumnCommand Logical Command for ANALYZE TABLE...COMPUTE STATISTICS FOR COLUMNS SQL Command

AnalyzeColumnCommand is a logical command for ANALYZE TABLE with FOR COLUMNS clause (and no PARTITION specification).

```
ANALYZE TABLE tableName COMPUTE STATISTICS FOR COLUMNS columnNames
```

```

// Make the example reproducible
val tableName = "t1"
import org.apache.spark.sql.catalyst.TableIdentifier
val tableId = TableIdentifier(tableName)

val sessionCatalog = spark.sessionState.catalog
sessionCatalog.dropTable(tableId, ignoreIfNotExists = true, purge = true)

val df = Seq((0, 0.0, "zero"), (1, 1.4, "one")).toDF("id", "p1", "p2")
df.write.saveAsTable("t1")

// AnalyzeColumnCommand represents ANALYZE TABLE...FOR COLUMNS SQL command
val allCols = df.columns.mkString(",")
val analyzeTableSQL = s"ANALYZE TABLE $tableName COMPUTE STATISTICS FOR COLUMNS $allCols"
val plan = spark.sql(analyzeTableSQL).queryExecution.logical
import org.apache.spark.sql.execution.command.AnalyzeColumnCommand
val cmd = plan.asInstanceOf[AnalyzeColumnCommand]
scala> println(cmd)
AnalyzeColumnCommand `t1`, [id, p1, p2]

spark.sql(analyzeTableSQL)
val stats = sessionCatalog.getTableMetadata(tableId).stats.get
scala> println(stats.simpleString)
1421 bytes, 2 rows

scala> stats.colStats.map { case (c, ss) => s"$c: $ss" }.foreach(println)
id: ColumnStat(2,Some(0),Some(1),0,4,4,None)
p1: ColumnStat(2,Some(0.0),Some(1.4),0,8,8,None)
p2: ColumnStat(2,None,None,0,4,4,None)

// Use DESC EXTENDED for friendlier output
scala> sql(s"DESC EXTENDED $tableName id").show
+-----+-----+
| info_name|info_value|
+-----+-----+
| col_name|      id|
| data_type|      int|
| comment|      NULL|
|       min|      0|
|       max|      1|
| num_nulls|      0|
|distinct_count|      2|
| avg_col_len|      4|
| max_col_len|      4|
| histogram|      NULL|
+-----+-----+

```

AnalyzeColumnCommand can generate column histograms when `spark.sql.statistics.histogram.enabled` configuration property is turned on (which is disabled by default). AnalyzeColumnCommand supports column histograms for the following data types:

- IntegralType
- DecimalType
- DoubleType
- FloatType
- DateType
- TimestampType

Note

Histograms can provide better estimation accuracy. Currently, Spark only supports equi-height histogram. Note that collecting histograms takes extra cost. For example, collecting column statistics usually takes only one table scan, but generating equi-height histogram will cause an extra table scan.

```
// ./bin/spark-shell --conf spark.sql.statistics.histogram.enabled=true
// Use the above example to set up the environment
// Make sure that ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS was run with histogram
enabled

// There are 254 bins by default
// Use spark.sql.statistics.histogram.numBins to control the bins
val descExtSQL = s"DESC EXTENDED $tableName p1"
scala> spark.sql(descExtSQL).show(truncate = false)
+-----+-----+
|info_name      |info_value
+-----+-----+
|col_name       |p1
|data_type      |double
|comment        |NULL
|min            |0.0
|max            |1.4
|num_nulls      |0
|distinct_count|2
|avg_col_len   |8
|max_col_len   |8
|histogram      |height: 0.007874015748031496, num_of_bins: 254
|bin_0          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_1          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_2          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_3          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_4          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_5          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_6          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_7          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_8          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
|bin_9          |lower_bound: 0.0, upper_bound: 0.0, distinct_count: 1|
+-----+-----+
only showing top 20 rows
```

Note	AnalyzeColumnCommand is described by analyze labeled alternative in statement expression in SqlBase.g4 and parsed using SparkSqlAstBuilder .
------	--

Note	AnalyzeColumnCommand is not supported on views.
------	---

Executing Logical Command — run Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note	run is part of RunnableCommand Contract to execute (run) a logical command.
------	---

run calculates the following statistics:

- sizeInBytes
- stats for each column

Caution	FIXME
---------	-------

Computing Statistics for Specified Columns — computeColumnStats Internal Method

```
computeColumnStats(  
    sparkSession: SparkSession,  
    tableIdent: TableIdentifier,  
    columnNames: Seq[String]): (Long, Map[String, ColumnStat])
```

computeColumnStats ...FIXME

Note	computeColumnStats is used exclusively when AnalyzeColumnCommand is executed.
------	---

computePercentiles Internal Method

```
computePercentiles(  
    attributesToAnalyze: Seq[Attribute],  
    sparkSession: SparkSession,  
    relation: LogicalPlan): AttributeMap[ArrayData]
```

computePercentiles ...FIXME

Note

`computePercentiles` is used exclusively when `AnalyzeColumnCommand` is executed (and computes column statistics).

Creating AnalyzeColumnCommand Instance

`AnalyzeColumnCommand` takes the following when created:

- `TableIdentifier`
- Column names

AnalyzePartitionCommand Logical Command — Computing Partition-Level Statistics (Total Size and Row Count)

`AnalyzePartitionCommand` is a [logical command](#) that [computes statistics](#) (i.e. [total size](#) and [row count](#)) for [table partitions](#) and stores the stats in a metastore.

`AnalyzePartitionCommand` is [created](#) exclusively for `ANALYZE TABLE` with `PARTITION` specification only (i.e. no `FOR COLUMNS` clause).

```
// Seq((0, 0, "zero"), (1, 1, "one")).toDF("id", "p1", "p2").write.partitionBy("p1", "p2").saveAsTable("t1")
val analyzeTable = "ANALYZE TABLE t1 PARTITION (p1, p2) COMPUTE STATISTICS"
val plan = spark.sql(analyzeTable).queryExecution.logical
import org.apache.spark.sql.execution.command.AnalyzePartitionCommand
val cmd = plan.asInstanceOf[AnalyzePartitionCommand]
scala> println(cmd)
AnalyzePartitionCommand `t1`, Map(p1 -> None, p2 -> None), false
```

Executing Logical Command (Computing Partition-Level Statistics and Altering Metastore) — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note	<code>run</code> is part of RunnableCommand Contract to execute (run) a logical command.
------	--

`run` requests the session-specific `sessionCatalog` for the [metadata](#) of the [table](#) and makes sure that it is not a view.

Note	<code>run</code> uses the input <code>SparkSession</code> to access the session-specific <code>SessionState</code> that in turn is used to access the current <code>SessionCatalog</code> .
------	---

`run` [getPartitionSpec](#).

`run` requests the session-specific `sessionCatalog` for the [partitions](#) per the partition specification.

`run` finishes when the table has no partitions defined in a metastore.

`run` [computes row count statistics per partition](#) unless `noscan` flag was enabled.

`run` calculates total size (in bytes) (aka *partition location size*) for every table partition and creates a CatalogStatistics with the current statistics if different from the statistics recorded in the metastore (with a new row count statistic computed earlier).

In the end, `run` alters table partition metadata for partitions with the statistics changed.

`run` reports a `NoSuchPartitionException` when partitions do not match the metastore.

`run` reports an `AnalysisException` when executed on a view.

```
ANALYZE TABLE is not supported on views.
```

Computing Row Count Statistics Per Partition — `calculateRowCountsPerPartition` Internal Method

```
calculateRowCountsPerPartition(  
    sparkSession: SparkSession,  
    tableMeta: CatalogTable,  
    partitionValueSpec: Option[TablePartitionSpec]): Map[TablePartitionSpec, BigInt]
```

`calculateRowCountsPerPartition` ...FIXME

Note	<code>calculateRowCountsPerPartition</code> is used exclusively when <code>AnalyzePartitionCommand</code> is executed .
------	---

getPartitionSpec Internal Method

```
getPartitionSpec(table: CatalogTable): Option[TablePartitionSpec]
```

`getPartitionSpec` ...FIXME

Note	<code>getPartitionSpec</code> is used exclusively when <code>AnalyzePartitionCommand</code> is executed .
------	---

Creating AnalyzePartitionCommand Instance

`AnalyzePartitionCommand` takes the following when created:

- `TableIdentifier`
- Partition specification

- `noscan` flag (enabled by default) that indicates whether `NOSCAN` option was used or not

AnalyzeTableCommand Logical Command — Computing Table-Level Statistics (Total Size and Row Count)

`AnalyzeTableCommand` is a [logical command](#) that [computes statistics](#) (i.e. [total size](#) and [row count](#)) for a [table](#) and stores the stats in a metastore.

`AnalyzeTableCommand` is [created](#) exclusively for `ANALYZE TABLE` with no `PARTITION` specification and `FOR COLUMNS` clause.

```
// Seq((0, 0, "zero"), (1, 1, "one")).toDF("id", "p1", "p2").write.partitionBy("p1", "p2").saveAsTable("t1")
val sqlText = "ANALYZE TABLE t1 COMPUTE STATISTICS NOSCAN"
val plan = spark.sql(sqlText).queryExecution.logical
import org.apache.spark.sql.execution.command.AnalyzeTableCommand
val cmd = plan.asInstanceOf[AnalyzeTableCommand]
scala> println(cmd)
AnalyzeTableCommand `t1`, false
```

Executing Logical Command (Computing Table-Level Statistics and Altering Metastore) — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note	<code>run</code> is part of RunnableCommand Contract to execute (run) a logical command.
------	--

`run` requests the session-specific `sessionCatalog` for the [metadata](#) of the [table](#) and makes sure that it is not a view (aka *temporary table*).

Note	<code>run</code> uses the input <code>sparkSession</code> to access the session-specific <code>SessionState</code> that in turn gives access to the current <code>SessionCatalog</code> .
------	---

`run` computes the [total size](#) and, without [NOSCAN](#) flag, the [row count](#) statistics of the table.

Note	<code>run</code> uses <code>SparkSession</code> to find the table in a metastore.
------	---

In the end, `run` [alters table statistics](#) if [different from the existing table statistics in metastore](#).

run throws a `AnalysisException` when executed on a view.

```
ANALYZE TABLE is not supported on views.
```

	Row count statistics triggers a Spark job to count the number of rows in a table (that is, with no <code>NOSCAN</code> flag).
Note	<pre>// Seq((0, 0, "zero"), (1, 1, "one")).toDF("id", "p1", "p2").write.partitionBy(3).insertInto("t1") val sqlText = "ANALYZE TABLE t1 COMPUTE STATISTICS" val plan = spark.sql(sqlText).queryExecution.logical import org.apache.spark.sql.execution.command.AnalyzeTableCommand val cmd = plan.asInstanceOf[AnalyzeTableCommand] scala> println(cmd) AnalyzeTableCommand `t1`, false // Execute ANALYZE TABLE // Check out web UI's Jobs tab for the number of Spark jobs // http://localhost:4040/jobs/ spark.sql(sqlText).show</pre>

Creating AnalyzeTableCommand Instance

`AnalyzeTableCommand` takes the following when created:

- `TableIdentifier`
- `noscan` flag (enabled by default) that indicates whether `NOSCAN` option was used or not

AppendData Logical Operator — Appending Data to DataSourceV2

`AppendData` is a [logical operator](#) that represents appending data (the result of executing a [structured query](#)) to a [table](#) (with the [columns matching by name or position](#)) in [DataSource API V2](#).

`AppendData` is [created](#) (indirectly via `byName` or `byPosition` factory methods) only for tests.

Note

`AppendData` has replaced the deprecated [WriteToDataSourceV2](#) logical operator.

`AppendData` takes the following to be created:

- `NamedRelation` for the table (to append data to)
- [Logical operator](#) (for the query)
- `isByName` flag

`AppendData` has a [single child logical operator](#) that is exactly the [logical operator](#).

`AppendData` is resolved using [ResolveOutputRelation](#) logical resolution rule.

`AppendData` is planned (*replaced*) to [WriteToDataSourceV2Exec](#) physical operator (when the [table](#) is a [DataSourceV2Relation](#) logical operator).

byName Factory Method

```
byName(table: NamedRelation, df: LogicalPlan): AppendData
```

`byName` simply creates a [AppendData](#) logical operator with the `isByName` flag on (`true`).

Note

`byName` seems used only for tests.

byPosition Factory Method

```
byPosition(table: NamedRelation, query: LogicalPlan): AppendData
```

`byPosition` simply creates a [AppendData](#) logical operator with the `isByName` flag off (`false`).

Note	byPosition seems used only for tests.
------	---------------------------------------

ClearCacheCommand Logical Command

`ClearCacheCommand` is a [logical command](#) to remove all cached tables from the in-memory cache.

`ClearCacheCommand` corresponds to `CLEAR CACHE` SQL statement.

Note	<code>clearCacheCommand</code> is described by <code>clearCache</code> labeled alternative in <code>statement expression</code> in <code>SqlBase.g4</code> and parsed using SparkSqlParser .
------	--

CreateDataSourceTableAsSelectCommand

Logical Command

`CreateDataSourceTableAsSelectCommand` is a [logical command](#) that [FIXME](#).

Executing Logical Command — `run` Method

```
run(session: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` ...[FIXME](#)

CreateDataSourceTableCommand Logical Command

`CreateDataSourceTableCommand` is a [logical command](#) that [creates a new table](#) (in a session-scoped `SessionCatalog`).

`CreateDataSourceTableCommand` is created exclusively when [DataSourceAnalysis](#) posthoc logical resolution rule resolves a [CreateTable](#) logical operator for a non-Hive table provider with no query.

`CreateDataSourceTableCommand` takes a `table` metadata and `ignoreIfExists` flag.

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` [creates a new table](#) in a session-scoped `SessionCatalog` .

Note

`run` uses the input `SparkSession` to [access SessionState](#) that in turn is used to [access the current SessionCatalog](#).

Internally, `run` [creates a BaseRelation](#) to access the table's schema.

Caution

FIXME

Note

`run` accepts tables only (not views) with the provider defined.

CreateHiveTableAsSelectCommand Logical Command

`CreateHiveTableAsSelectCommand` is a logical command that [FIXME](#).

Executing Logical Command — `run` Method

```
run(session: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` ...[FIXME](#)

CreateTable Logical Operator

`CreateTable` is a [logical operator](#) that represents (is [created](#) for) the following:

- `DataFrameWriter` is requested to [createTable](#) (when requested to [saveAsTable](#))
- `SparkSqlAstBuilder` is requested to [visitCreateTable](#) (for `CREATE TABLE` SQL command) or [visitCreateHiveTable](#) (for `CREATE EXTERNAL TABLE` SQL command)
- `CatalogImpl` is requested to [create a table](#)

`CreateTable` requires that the [table provider](#) of the [CatalogTable](#) is defined or throws an `AssertionError`:

```
assertion failed: The table to be created must have a provider.
```

`CreateTable` can never be [resolved](#) and is replaced (*resolved*) with a logical command at analysis phase in the following rules:

- (for non-hive data source tables) [DataSourceAnalysis](#) posthoc logical resolution rule to a [CreateDataSourceTableCommand](#) or a [CreateDataSourceTableAsSelectCommand](#) logical command (when the `query` was defined or not, respectively)
- (for hive tables) [HiveAnalysis](#) post-hoc logical resolution rule to a [CreateTableCommand](#) or a [CreateHiveTableAsSelectCommand](#) logical command (when `query` was defined or not, respectively)

Creating CreateTable Instance

`CreateTable` takes the following when created:

- [Table metadata](#)
- [SaveMode](#)
- [Logical query plan](#)

`CreateTable` initializes the [internal registries and counters](#).

CreateTableCommand Logical Command

`CreateTableCommand` is a [logical command](#) that [FIXME](#).

Executing Logical Command — `run` Method

```
run(session: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` ...[FIXME](#)

CreateTempViewUsing Logical Command

`CreateTempViewUsing` is a [logical command](#) for [creating or replacing a temporary view](#) (global or not) using a [data source](#).

`CreateTempViewUsing` is [created](#) to represent `CREATE TEMPORARY VIEW ... USING SQL` statements.

```
val sqlText = s"""
|CREATE GLOBAL TEMPORARY VIEW myTempCsvView
|(id LONG, name STRING)
|USING csv
""".stripMargin
// Logical commands are executed at analysis
scala> sql(sqlText)
res4: org.apache.spark.sql.DataFrame = []

scala> spark.catalog.listTables(spark.sharedState.globalTempViewManager.database).show
+-----+-----+-----+-----+
|      name| database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|mytempcsvview|global_temp|      null|TEMPORARY|     true|
+-----+-----+-----+-----+
```

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` creates a [DataSource](#) and requests it to [resolve itself](#) (i.e. create a [BaseRelation](#)).

`run` then requests the input `SparkSession` to [create a DataFrame](#) from the [BaseRelation](#) that is used to [get the analyzed logical plan](#) (that is the view definition of the temporary table).

Depending on the `global` flag, `run` requests the `SessionCatalog` to [createGlobalTempView](#) (`global` flag is on) or [createTempView](#) (`global` flag is off).

`run` throws an `AnalysisException` when executed with `hive` provider.

```
Hive data source can only be used with tables, you can't use it with CREATE TEMP VIEW  
USING
```

Creating CreateTempViewUsing Instance

`CreateTempViewUsing` takes the following when created:

- `TableIdentifier`
- Optional user-defined schema (as `StructType`)
- `replace` flag
- `global` flag
- Name of the `data source provider`
- Options (as `Map[String, String]`)

`argString` Method

```
argString: String
```

Note

`argString` is part of the [TreeNode Contract](#) to...FIXME.

`argString` ...FIXME

CreateViewCommand Logical Command

`CreateViewCommand` is a [logical command](#) for creating or replacing a view or a table.

`CreateViewCommand` is [created](#) to represent the following:

- [CREATE VIEW AS](#) SQL statements
- `Dataset` operators: [Dataset.createTempView](#), [Dataset.createOrReplaceTempView](#), [Dataset.createGlobalTempView](#) and [Dataset.createOrReplaceGlobalTempView](#)

Caution

FIXME What's the difference between `createTempViewUsing` ?

`CreateViewCommand` works with different [view types](#).

Table 1. CreateViewCommand Behaviour Per View Type

View Type	Description / Side Effect
<code>LocalTempView</code>	A session-scoped local temporary view that is available until the session, that has created it, is stopped. When executed, <code>CreateViewCommand</code> requests the current SessionCatalog to create a temporary view.
<code>GlobalTempView</code>	A cross-session global temporary view that is available until the Spark application stops. When executed, <code>CreateViewCommand</code> requests the current SessionCatalog to create a global view.
<code>PersistedView</code>	A cross-session persisted view that is available until dropped. When executed, <code>CreateViewCommand</code> checks if the table exists. If it does and replace is enabled <code>CreateViewCommand</code> requests the current SessionCatalog to alter a table. Otherwise, when the table does not exist, <code>CreateViewCommand</code> requests the current SessionCatalog to create it.

```

/* CREATE [OR REPLACE] [[GLOBAL]] TEMPORARY]
VIEW [IF NOT EXISTS] tableIdentifier
[identifierCommentList] [COMMENT STRING]
[PARTITIONED ON identifierList]
[TBLPROPERTIES tablePropertyList] AS query */

// Demo table for "AS query" part
spark.range(10).write.mode("overwrite").saveAsTable("t1")

```

```

// The "AS" query
val asQuery = "SELECT * FROM t1"

// The following queries should all work fine
val q1 = "CREATE VIEW v1 AS " + asQuery
sql(q1)

val q2 = "CREATE OR REPLACE VIEW v1 AS " + asQuery
sql(q2)

val q3 = "CREATE OR REPLACE TEMPORARY VIEW v1 " + asQuery
sql(q3)

val q4 = "CREATE OR REPLACE GLOBAL TEMPORARY VIEW v1 " + asQuery
sql(q4)

val q5 = "CREATE VIEW IF NOT EXISTS v1 AS " + asQuery
sql(q5)

// The following queries should all fail
// the number of user-specified columns does not match the schema of the AS query
val qf1 = "CREATE VIEW v1 (c1 COMMENT 'comment', c2) AS " + asQuery
scala> sql(qf1)
org.apache.spark.sql.AnalysisException: The number of columns produced by the SELECT clause (num: `1`) does not match the number of column names specified by CREATE VIEW (num: `2`).;
    at org.apache.spark.sql.execution.command.CreateViewCommand.run(views.scala:134)
    at org.apache.spark.sql.execution.command.ExecutedCommandExec.sideEffectResult$lzycompute(commands.scala:70)
    at org.apache.spark.sql.execution.command.ExecutedCommandExec.sideEffectResult(commands.scala:68)
    at org.apache.spark.sql.execution.command.ExecutedCommandExec.executeCollect(commands.scala:79)
    at org.apache.spark.sql.Dataset$$anonfun$6.apply(Dataset.scala:190)
    at org.apache.spark.sql.Dataset$$anonfun$6.apply(Dataset.scala:190)
    at org.apache.spark.sql.Dataset$$anonfun$52.apply(Dataset.scala:3254)
    at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.scala:77)
    at org.apache.spark.sql.Dataset.withAction(Dataset.scala:3253)
    at org.apache.spark.sql.Dataset.<init>(Dataset.scala:190)
    at org.apache.spark.sql.Dataset$.ofRows(Dataset.scala:75)
    at org.apache.spark.sql.SparkSession.sql(SparkSession.scala:641)
    ... 49 elided

// CREATE VIEW ... PARTITIONED ON is not allowed
val qf2 = "CREATE VIEW v1 PARTITIONED ON (c1, c2) AS " + asQuery
scala> sql(qf2)
org.apache.spark.sql.catalyst.parser.ParseException:
Operation not allowed: CREATE VIEW ... PARTITIONED ON(line 1, pos 0)

// Use the same name of t1 for a new view
val qf3 = "CREATE VIEW t1 AS " + asQuery

```

```

scala> sql(qf3)
org.apache.spark.sql.AnalysisException: `t1` is not a view;
   at org.apache.spark.sql.execution.command.CreateViewCommand.run(views.scala:156)
   at org.apache.spark.sql.execution.command.ExecutedCommandExec.sideEffectResult$lzyco
mpute(commands.scala:70)
   at org.apache.spark.sql.execution.command.ExecutedCommandExec.sideEffectResult(comma
nds.scala:68)
   at org.apache.spark.sql.execution.command.ExecutedCommandExec.executeCollect(command
s.scala:79)
   at org.apache.spark.sql.Dataset$$anonfun$6.apply(Dataset.scala:190)
   at org.apache.spark.sql.Dataset$$anonfun$6.apply(Dataset.scala:190)
   at org.apache.spark.sql.Dataset$$anonfun$52.apply(Dataset.scala:3254)
   at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.sc
a:77)
   at org.apache.spark.sql.Dataset.withAction(Dataset.scala:3253)
   at org.apache.spark.sql.Dataset.<init>(Dataset.scala:190)
   at org.apache.spark.sql.Dataset$.ofRows(Dataset.scala:75)
   at org.apache.spark.sql.SparkSession.sql(SparkSession.scala:641)
... 49 elided

// View already exists
val qf4 = "CREATE VIEW v1 AS " + asQuery
scala> sql(qf4)
org.apache.spark.sql.AnalysisException: View `v1` already exists. If you want to updat
e the view definition, please use ALTER VIEW AS or CREATE OR REPLACE VIEW AS;
   at org.apache.spark.sql.execution.command.CreateViewCommand.run(views.scala:169)
   at org.apache.spark.sql.execution.command.ExecutedCommandExec.sideEffectResult$lzyco
mpute(commands.scala:70)
   at org.apache.spark.sql.execution.command.ExecutedCommandExec.sideEffectResult(comma
nds.scala:68)
   at org.apache.spark.sql.execution.command.ExecutedCommandExec.executeCollect(command
s.scala:79)
   at org.apache.spark.sql.Dataset$$anonfun$6.apply(Dataset.scala:190)
   at org.apache.spark.sql.Dataset$$anonfun$6.apply(Dataset.scala:190)
   at org.apache.spark.sql.Dataset$$anonfun$52.apply(Dataset.scala:3254)
   at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.sc
a:77)
   at org.apache.spark.sql.Dataset.withAction(Dataset.scala:3253)
   at org.apache.spark.sql.Dataset.<init>(Dataset.scala:190)
   at org.apache.spark.sql.Dataset$.ofRows(Dataset.scala:75)
   at org.apache.spark.sql.SparkSession.sql(SparkSession.scala:641)
... 49 elided

```

`CreateViewCommand` returns the [child logical query plan](#) when requested for the inner nodes (that should be shown as an inner nested tree of this node).

```

val sqlText = "CREATE VIEW v1 AS " + asQuery
val plan = spark.sessionState.sqlParser.parsePlan(sqlText)
scala> println(plan.numberedTreeString)
00 CreateViewCommand `v1`, SELECT * FROM t1, false, false, PersistedView
01   +- 'Project [*]
02     +- 'UnresolvedRelation `t1`'

```

Creating CatalogTable — `prepareTable` Internal Method

```
prepareTable(session: SparkSession, analyzedPlan: LogicalPlan): CatalogTable
```

`prepareTable` ...FIXME

Note	<code>prepareTable</code> is used exclusively when <code>CreateViewCommand</code> logical command is executed .
------	---

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note	<code>run</code> is part of RunnableCommand Contract to execute (run) a logical command.
------	--

`run` requests the input `SparkSession` for the `SessionState` that is in turn requested to [execute](#) the child logical plan (which simply creates a `QueryExecution`).

Note	<code>run</code> uses a common idiom in Spark SQL to make sure that a logical plan can be analyzed, i.e.
------	--

Note	<pre> val qe = sparkSession.sessionState.executePlan(child) qe.assertAnalyzed() val analyzedPlan = qe.analyzed </pre>
------	---

`run` [verifyTemporaryObjectsNotExists](#).

`run` requests the input `SparkSession` for the `SessionState` that is in turn requested for the [SessionCatalog](#).

`run` then branches off per the `ViewType`:

- For [local temporary views](#), `run` [alias](#) the analyzed plan and requests the `SessionCatalog` to [create or replace a local temporary view](#)

- For [global temporary views](#), `run` also [alias](#) the analyzed plan and requests the `SessionCatalog` to [create or replace a global temporary view](#)
- For [persisted views](#), `run` asks the `SessionCatalog` whether the [table exists or not](#) (given [TableIdentifier](#)).
 - If the [table](#) exists and the [allowExisting](#) flag is on, `run` simply does nothing (and exits)
 - If the [table](#) exists and the [replace](#) flag is on, `run` requests the `SessionCatalog` for the [table metadata](#) and replaces the table, i.e. `run` requests the `SessionCatalog` to [drop the table](#) followed by [re-creating it](#) (with a [new CatalogTable](#))
 - If however the [table](#) does not exist, `run` simply requests the `SessionCatalog` to [create it](#) (with a [new CatalogTable](#))

`run` throws an `AnalysisException` for [persisted views](#) when they already exist, the [allowExisting](#) flag is off and the table type is not a view.

[name] is not a view

`run` throws an `AnalysisException` for [persisted views](#) when they already exist and the [allowExisting](#) and [replace](#) flags are off.

View [name] already exists. If you want to update the view definition, please use ALTER VIEW AS or CREATE OR REPLACE VIEW AS

`run` throws an `AnalysisException` if the [userSpecifiedColumns](#) are defined and their numbers is different from the number of [output schema attributes](#) of the analyzed logical plan.

The number of columns produced by the SELECT clause (num: `'[output.length]`) does not match the number of column names specified by CREATE VIEW (num: `'[userSpecifiedColumns.length]`).

Creating CreateViewCommand Instance

`CreateViewCommand` takes the following when created:

- `TableIdentifier`
- User-defined columns (as `Seq[(String, Option[String])]`)
- Optional comment

- Properties (as `Map[String, String]`)
- Optional DDL statement
- Child [logical plan](#)
- `allowExisting` flag
- `replace` flag
- [ViewType](#)

verifyTemporaryObjectsNotExists Internal Method

```
verifyTemporaryObjectsNotExists(sparkSession: SparkSession): Unit
```

`verifyTemporaryObjectsNotExists` ...FIXME

Note	<code>verifyTemporaryObjectsNotExists</code> is used exclusively when <code>CreateViewCommand</code> logical command is executed .
------	--

aliasPlan Internal Method

```
aliasPlan(session: SparkSession, analyzedPlan: LogicalPlan): LogicalPlan
```

`aliasPlan` ...FIXME

Note	<code>aliasPlan</code> is used when <code>CreateViewCommand</code> logical command is executed (and prepareTable).
------	---

DataSourceV2Relation Leaf Logical Operator

`DataSourceV2Relation` is a [leaf logical operator](#) that represents a data scan (*data reading*) or data writing in the [Data Source API V2](#).

`DataSourceV2Relation` is [created](#) (indirectly via `create` helper method) exclusively when `DataFrameReader` is requested to "load" data (as a `DataFrame`) (from a data source with [ReadSupport](#)).

`DataSourceV2Relation` takes the following to be created:

- `DataSourceV2`
- `Output attributes` (`Seq[AttributeReference]`)
- `Options` (`Map[String, String]`)
- Optional `TableIdentifier` (default: undefined, i.e. `None`)
- User-defined `schema` (default: undefined, i.e. `None`)

When used to represent a data scan (*data reading*), `DataSourceV2Relation` is planned (*translated*) to a [ProjectExec](#) with a [DataSourceV2ScanExec](#) physical operator (possibly under the [FilterExec](#) operator) when [DataSourceV2Strategy](#) execution planning strategy is requested to [plan a logical plan](#).

When used to represent a data write (with [AppendData](#) logical operator), `DataSourceV2Relation` is planned (*translated*) to a [WriteToDataSourceV2Exec](#) physical operator (with the [DataSourceWriter](#)) when [DataSourceV2Strategy](#) execution planning strategy is requested to [plan a logical plan](#).

`DataSourceV2Relation` object defines a `SourceHelpers` implicit class that extends `DataSourceV2` instances with the additional [extension methods](#).

Creating DataSourceV2Relation Instance — `create` Factory Method

```
create(
  source: DataSourceV2,
  options: Map[String, String],
  tableIdent: Option[TableIdentifier] = None,
  userSpecifiedSchema: Option[StructType] = None): DataSourceV2Relation
```

`create` requests the given [DataSourceV2](#) to create a [DataSourceReader](#) (with the given options and user-specified schema).

`create` finds the table in the given options unless the optional `tableIdent` is defined.

In the end, `create` creates a [DataSourceV2Relation](#).

Note

`create` is used exclusively when `DataFrameReader` is requested to "load" data (as a [DataFrame](#)) (from a data source with [ReadSupport](#)).

Computing Statistics — `computeStats` Method

```
computeStats(): Statistics
```

Note

`computeStats` is part of the [LeafNode Contract](#) to compute a [Statistics](#).

`computeStats` ...FIXME

Creating DataSourceReader — `newReader` Method

```
newReader(): DataSourceReader
```

`newReader` simply requests (*delegates to*) the [DataSourceV2](#) to create a [DataSourceReader](#).

Note

`DataSourceV2Relation` object defines the [SourceHelpers](#) implicit class to "extend" the marker [DataSourceV2](#) type with the method to create a [DataSourceReader](#).

Note

`newReader` is used when:

- `DataSourceV2Relation` is requested to `computeStats`
- `DataSourceV2Strategy` execution planning strategy is requested to `plan` a [DataSourceV2Relation logical operator](#)

Creating DataSourceWriter — `newWriter` Method

```
newWriter(): DataSourceWriter
```

`newWriter` simply requests (*delegates to*) the [DataSourceV2](#) to create a [DataSourceWriter](#).

Note	<code>DataSourceV2Relation</code> object defines the SourceHelpers implicit class to "extend" the marker DataSourceV2 type with the method to create a DataSourceWriter .
Note	<code>newWriter</code> is used exclusively when <code>DataSourceV2Strategy</code> execution planning strategy is requested to plan an AppendData logical operator .

SourceHelpers Implicit Class

`DataSourceV2Relation` object defines a `SourceHelpers` implicit class that extends [DataSourceV2](#) instances with the additional [extension methods](#).

Table 1. SourceHelpers' Extension Methods

Method	Description
asReadSupport	<pre>asReadSupport: ReadSupport</pre> <p>Used exclusively for createReader implicit method</p>
asWriteSupport	<pre>asWriteSupport: WriteSupport</pre> <p>Used when...FIXME</p>
name	<pre>name: String</pre> <p>Used when...FIXME</p>
createReader	<pre>createReader(options: Map[String, String], userSpecifiedSchema: Option[StructType]): DataSourceReader</pre> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataSourceV2Relation</code> logical operator is requested to create a DataSourceReader • <code>DataSourceV2Relation</code> factory object is requested to create a DataSourceV2Relation (when <code>DataFrameReader</code> is requested to "load" data (as a <code>DataFrame</code>) from a data source with ReadSupport)
createWriter	<pre>createWriter(options: Map[String, String], schema: StructType): DataSourceWriter</pre> <p>Creates a DataSourceWriter</p> <p>Used when...FIXME</p>
Tip	Read up on implicit classes in the official documentation of the Scala programming language .

DescribeColumnCommand Logical Command for DESCRIBE TABLE SQL Command with Column

`DescribeColumnCommand` is a logical command for `DESCRIBE TABLE` SQL command with a single column only (i.e. no `PARTITION` specification).

```
[DESC|DESCRIBE] TABLE? [EXTENDED|FORMATTED] table_name column_name

// Make the example reproducible
val tableName = "t1"
import org.apache.spark.sql.catalyst.TableIdentifier
val tableId = TableIdentifier(tableName)

val sessionCatalog = spark.sessionState.catalog
sessionCatalog.dropTable(tableId, ignoreIfNotExists = true, purge = true)

val df = Seq((0, 0.0, "zero"), (1, 1.4, "one")).toDF("id", "p1", "p2")
df.write.saveAsTable("t1")

// DescribeColumnCommand represents DESC EXTENDED tableName colName SQL command
val descExtSQL = "DESC EXTENDED t1 p1"
val plan = spark.sql(descExtSQL).queryExecution.logical
import org.apache.spark.sql.execution.command.DescribeColumnCommand
val cmd = plan.asInstanceOf[DescribeColumnCommand]
scala> println(cmd)
DescribeColumnCommand `t1`, [p1], true

scala> spark.sql(descExtSQL).show
+-----+-----+
| info_name|info_value|
+-----+-----+
|   col_name|      p1|
|  data_type|    double|
|   comment|      NULL|
|      min|      NULL|
|      max|      NULL|
| num_nulls|      NULL|
|distinct_count|      NULL|
| avg_col_len|      NULL|
| max_col_len|      NULL|
| histogram|      NULL|
+-----+-----+

// Run ANALYZE TABLE...FOR COLUMNS SQL command to compute the column statistics
val allCols = df.columns.mkString(",")
```

```

val analyzeTableSQL = s"ANALYZE TABLE $tableName COMPUTE STATISTICS FOR COLUMNS $allColumns"
spark.sql(analyzeTableSQL)

scala> spark.sql(descExtSQL).show
+-----+-----+
| info_name|info_value|
+-----+-----+
| col_name|      p1|
| data_type|    double|
| comment|      NULL|
| min|      0.0|
| max|      1.4|
| num_nulls|      0|
| distinct_count|      2|
| avg_col_len|      8|
| max_col_len|      8|
| histogram|      NULL|
+-----+-----+

```

`DescribeColumnCommand` defines the [output schema](#) with the following columns:

- `info_name` with "name of the column info" comment
- `info_value` with "value of the column info" comment

Note	<code>DescribeColumnCommand</code> is described by <code>describeTable</code> labeled alternative in statement expression in <code>SqlBase.g4</code> and parsed using SparkSqlParser .
------	--

Executing Logical Command (Describing Column with Optional Statistics) — `run` Method

```
run(session: SparkSession): Seq[Row]
```

Note	<code>run</code> is part of RunnableCommand Contract to execute (run) a logical command.
------	--

`run` resolves the [column name](#) in `table` and makes sure that it is a "flat" field (i.e. not of a nested data type).

`run` requests the `SessionCatalog` for the [table metadata](#).

Note	<code>run</code> uses the input <code>SparkSession</code> to access <code>SessionState</code> that in turn is used to access the <code>SessionCatalog</code> .
------	--

`run` takes the [column statistics](#) from the [table statistics](#) if available.

Note

`Column statistics` are available (in the `table statistics`) only after `ANALYZE TABLE FOR COLUMNS` SQL command was run.

`run adds comment metadata if available for the column.`

`run gives the following rows (in that order):`

1. `col_name`
2. `data_type`
3. `comment`

If `DescribeColumnCommand` command was executed with `EXTENDED` or `FORMATTED` option, `run` gives the following additional rows (in that order):

1. `min`
2. `max`
3. `num_nulls`
4. `distinct_count`
5. `avg_col_len`
6. `max_col_len`
7. `histogram`

`run gives NULL for the value of the comment and statistics if not available.`

histogramDescription Internal Method

```
histogramDescription(histogram: Histogram): Seq[Row]
```

`histogramDescription ...FIXME`

Note

`histogramDescription` is used exclusively when `DescribeColumnCommand` is executed with `EXTENDED` or `FORMATTED` option turned on.

Creating DescribeColumnCommand Instance

`DescribeColumnCommand` takes the following when created:

- `TableIdentifier`

- Column name
- `isExtended` flag that indicates whether EXTENDED or FORMATTED option was used or not

DescribeTableCommand Logical Command

`DescribeTableCommand` is a [logical command](#) that [executes](#) a `DESCRIBE TABLE` SQL statement.

`DescribeTableCommand` is [created](#) exclusively when `sparkSqlAstBuilder` is requested to parse `DESCRIBE TABLE` SQL statement (with no column specified).

`DescribeTableCommand` uses the following [output schema](#):

- `col_name` as the name of the column
- `data_type` as the data type of the column
- `comment` as the comment of the column

```
spark.range(1).createOrReplaceTempView("demo")

// DESC view
scala> sql("DESC EXTENDED demo").show
+-----+-----+
|col_name|data_type|comment|
+-----+-----+
|      id|    bigint|    null|
+-----+-----+


// DESC table
// Make the demo reproducible
spark.sharedState.externalCatalog.dropTable(
  db = "default",
  table = "bucketed",
  ignoreIfNotExists = true,
  purge = true)
spark.range(10).write.bucketBy(5, "id").saveAsTable("bucketed")
assert(spark.catalog.tableExists("bucketed"))

// EXTENDED to include Detailed Table Information
// Note no partitions used
// Could also be FORMATTED
scala> sql("DESC EXTENDED bucketed").show(numRows = 50, truncate = false)
+-----+
-----+-----+
|col_name          |data_type
|comment          |
+-----+
-----+-----+
|id              |bigint
|null           |
|
```

```

| # Detailed Table Information |
| Database           | default
| Table              | bucketed
| Owner              | jacek
| Created Time       | Sun Sep 30 20:57:22 CEST 2018
| Last Access        | Thu Jan 01 01:00:00 CET 1970
| Created By         | Spark 2.3.1
| Type               | MANAGED
| Provider            | parquet
| Num Buckets        | 5
| Bucket Columns      | [ `id` ]
| Sort Columns        | []
| Table Properties    | [[transient_lastDdlTime=1538333842]]
| Statistics          | 3740 bytes
| Location            | file:/Users/jacek/dev/apps/spark-2.3.1-bin-hadoop2.7/spa
rk-warehouse/bucketed
| Serde Library       | org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
| InputFormat          | org.apache.hadoop.mapred.SequenceFileInputFormat
| OutputFormat         | org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
| Storage Properties   | [[serialization.format=1]]
+-----+
-----+-----+
// Make the demo reproducible
val tableName = "partitioned_bucketed_sorted"
val partCol = "part"
spark.sharedState.externalCatalog.dropTable(
  db = "default",
  table = tableName,
  ignoreIfNotExists = true,
  purge = true)
spark.range(10)
  .withColumn("part", $"id" % 2) // extra column for partitions
  .write

```

```

.partitionBy(partCol)
.bucketBy(5, "id")
.sortBy("id")
.saveAsTable(tableName)
assert(spark.catalog.tableExists(tableName))
scala> sql(s"DESC EXTENDED $tableName").show(numRows = 50, truncate = false)
+-----+
-----+-----+
|col_name          |data_type
|comment|
+-----+-----+
-----+-----+
|id                |bigint
|                  |null   |
|part              |bigint
|                  |null   |
|# Partition Information |      |
|# col_name          |data_type
|comment|
|part              |bigint
|                  |null   |
|                  |      |
|# Detailed Table Information|      |
|Database          |default
|                  |      |
|Table             |partitioned_bucketed_sorted
|                  |      |
|Owner             |jacek
|                  |      |
|Created Time     |Mon Oct 01 10:05:32 CEST 2018
|                  |      |
|Last Access       |Thu Jan 01 01:00:00 CET 1970
|                  |      |
|Created By        |Spark 2.3.1
|                  |      |
|Type              |MANAGED
|                  |      |
|Provider           |parquet
|                  |      |
|Num Buckets       |5
|                  |      |
|Bucket Columns    |[`id`]
|                  |      |
|Sort Columns       |[`id`]
|                  |      |
|Table Properties   |[[transient_lastDdlTime=1538381132]]
|                  |      |
|Location           |file:/Users/jacek/dev/apps/spark-2.3.1-bin-hadoop2.7/spark-warehouse/partitioned_bucketed_sorted|
|                  |      |
|Serde Library      |org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

```

```

|InputFormat           |          |
|InputFormat           |org.apache.hadoop.mapred.SequenceFileInputFormat
|OutputFormat          |          |
|OutputFormat          |org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat
|Storage Properties   |          |
|Storage Properties   |[[serialization.format=1]]
|Partition Provider   |          |
|Partition Provider   |Catalog
+-----+
-----+-----+
scala> sql(s"DESCRIBE EXTENDED $tableName PARTITION ($partCol=1)").show(numRows = 50,
truncate = false)
+-----+-----+
-----+-----+
|col_name            |data_type
|comment             |
+-----+-----+
-----+-----+
|id                  |bigint
|null                |
|part                |bigint
|null                |
|# Partition Information | |
|comment             |
|# col_name           |data_type
|comment             |
|part                |bigint
|null                |
|                         |
|                         |
|# Detailed Partition Information|
|Database             |default
|Table                |partitioned_bucketed_sorted
|Partition Values     |[[part=1]]
|Location              |file:/Users/jacek/dev/apps/spark-2.3.1-bin-hadoop2.7
/spark-warehouse/partitioned_bucketed_sorted/part=1
|Serde Library         |org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
|InputFormat            |org.apache.hadoop.mapred.SequenceFileInputFormat
|OutputFormat           |org.apache.hadoop.hive ql.io.HiveSequenceFileOutputFormat
|Storage Properties    |[[path=file:/Users/jacek/dev/apps/spark-2.3.1-bin-hadoop2.7/spark-warehouse/partitioned_bucketed_sorted, serialization.format=1]]
|Partition Parameters  |{{totalSize=1870, numFiles=5, transient_lastDdlTime=1538381329}}
|Partition Statistics  |1870 bytes

```

# Storage Information			
Num Buckets	5		
Bucket Columns	[`id`]		
Sort Columns	[`id`]		
Location	file:/Users/jacek/dev/apps/spark-2.3.1-bin-hadoop2.7		
/spark-warehouse/partitioned_bucketed_sorted			
Serde Library	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe		
InputFormat	org.apache.hadoop.mapred.SequenceFileInputFormat		
OutputFormat	org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputF		
ormat			
Storage Properties	[[serialization.format=1]]		
+-----+-----+-----+-----+			
- - - - -			

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note

`run` is part of the [RunnableCommand Contract](#) to execute (run) a logical command.

`run` uses the [SessionCatalog](#) (of the [SessionState](#) of the input [SparkSession](#)) and branches off per the type of the table to display.

For a [temporary view](#), `run` requests the [SessionCatalog](#) to [lookupRelation](#) to access the [schema](#) and [describeSchema](#).

For all other table types, `run` does the following:

1. Requests the [SessionCatalog](#) to retrieve the table metadata from the external catalog ([metastore](#)) (as a [CatalogTable](#)) and [describeSchema](#) (with the [schema](#))
2. [describePartitionInfo](#)
3. [describeDetailedPartitionInfo](#) if the [TablePartitionSpec](#) is available or [describeFormattedTableInfo](#) when [isExtended](#) flag is on

Describing Detailed Partition and Storage Information

— `describeFormattedDetailedPartitionInfo` Internal Method

```
describeFormattedDetailedPartitionInfo(
    tableIdentifier: TableIdentifier,
    table: CatalogTable,
    partition: CatalogTablePartition,
    buffer: ArrayBuffer[Row]): Unit
```

`describeFormattedDetailedPartitionInfo` simply adds the following entries (rows) to the input mutable buffer:

1. A new line
2. **# Detailed Partition Information**
3. **Database** with the [database](#) of the given `table`
4. **Table** with the table of the given `tableIdentifier`
5. **Partition specification** (of the [CatalogTablePartition](#))
6. A new line
7. **# Storage Information**
8. Bucketing specification of the `table` (if defined)
9. Storage specification of the `table`

Note

`describeFormattedDetailedPartitionInfo` is used exclusively when `DescribeTableCommand` is requested to `describeDetailedPartitionInfo` with a non-empty `partitionSpec` and the `isExtended` flag on.

Describing Detailed Table Information

— `describeFormattedTableInfo` Internal Method

```
describeFormattedTableInfo(table: CatalogTable, buffer: ArrayBuffer[Row]): Unit
```

`describeFormattedTableInfo` ...FIXME

Note

`describeFormattedTableInfo` is used exclusively when `DescribeTableCommand` is requested to `run` for a [non-temporary table](#) and the `isExtended` flag on.

describeDetailedPartitionInfo Internal Method

```
describeDetailedPartitionInfo(
    tableIdentifier: TableIdentifier,
    table: CatalogTable,
    partition: CatalogTablePartition,
    buffer: ArrayBuffer[Row]): Unit
```

describeDetailedPartitionInfo ...FIXME

Note

`describeDetailedPartitionInfo` is used exclusively when `DescribeTableCommand` is requested to run with a non-empty `partitionSpec`.

Creating DescribeTableCommand Instance

`DescribeTableCommand` takes the following when created:

- `TableIdentifier`
- `TablePartitionSpec`
- `isExtended` flag

`DescribeTableCommand` initializes the [internal registries and counters](#).

describeSchema Internal Method

```
describeSchema(
    schema: StructType,
    buffer: ArrayBuffer[Row],
    header: Boolean): Unit
```

describeSchema ...FIXME

Note

`describeSchema` is used when...FIXME

Describing Partition Information**— describePartitionInfo Internal Method**

```
describePartitionInfo(table: CatalogTable, buffer: ArrayBuffer[Row]): Unit
```

describePartitionInfo ...FIXME

Note

`describePartitionInfo` is used when...FIXME

DeserializeToObject Unary Logical Operator

```
case class DeserializeToObject(  
    deserializer: Expression,  
    outputObjAttr: Attribute,  
    child: LogicalPlan) extends UnaryNode with ObjectProducer
```

`DeserializeToObject` is a [unary logical operator](#) that takes the input row from the input `child` [logical plan](#) and turns it into the input `outputObjAttr` [attribute](#) using the given `deserializer` [expression](#).

`DeserializeToObject` is a `ObjectProducer` which produces domain objects as output. `DeserializeToObject`'s output is a single-field safe row containing the produced object.

Note

`DeserializeToObject` is the result of [CatalystSerde.deserialize](#).

DropTableCommand Logical Command

`DropTableCommand` is a [logical command](#) for [FIXME](#).

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run ...FIXME`

Except

Except is...FIXME

Expand Unary Logical Operator

`Expand` is a [unary logical operator](#) that represents `Cube`, `Rollup`, `GroupingSets` and `TimeWindow` logical operators after they have been resolved at [analysis phase](#).

```
FIXME Examples for
1. Cube
2. Rollup
3. GroupingSets
4. See TimeWindow

val q = ...

scala> println(q.queryExecution.logical.numberedTreeString)
...
```

Note	<code>Expand</code> logical operator is resolved to ExpandExec physical operator in BasicOperators execution planning strategy.
------	---

Table 1. Expand's Properties

Name	Description
references	<code>AttributeSet</code> from projections
validConstraints	Empty set of expressions

Analysis Phase

`Expand` logical operator is resolved to at [analysis phase](#) in the following logical evaluation rules:

- [ResolveGroupingAnalytics](#) (for `Cube`, `Rollup`, `GroupingSets` logical operators)
- [TimeWindowing](#) (for `TimeWindow` logical operator)

Note	Aggregate → (Cube Rollup GroupingSets) → constructAggregate → constructExpand
------	---

```
val spark: SparkSession = ...
// using q from the example above
val plan = q.queryExecution.logical

scala> println(plan.numberedTreeString)
...FIXME
```

Rule-Based Logical Query Optimization Phase

- [ColumnPruning](#)
- [FoldablePropagation](#)
- [RewriteDistinctAggregates](#)

Creating Expand Instance

`Expand` takes the following when created:

- Projection [expressions](#)
- Output schema [attributes](#)
- Child [logical plan](#)

ExplainCommand Logical Command

`ExplainCommand` is a [logical command](#) with side effect that allows users to see how a structured query is structured and will eventually be executed, i.e. shows logical and physical plans with or without details about codegen and cost statistics.

When [executed](#), `ExplainCommand` computes a `QueryExecution` that is then used to output a single-column `DataFrame` with the following:

- **codegen explain**, i.e. [WholeStageCodegen](#) subtrees if `codegen` flag is enabled.
- **extended explain**, i.e. the parsed, analyzed, optimized logical plans with the physical plan if [extended](#) flag is enabled.
- **cost explain**, i.e. [optimized logical plan](#) with stats if `cost` flag is enabled.
- **simple explain**, i.e. the physical plan only when no `codegen` and `extended` flags are enabled.

`ExplainCommand` is created by Dataset's `explain` operator and `EXPLAIN` SQL statement (accepting `EXTENDED` and `CODEGEN` options).

```
// Explain in SQL

scala> sql("EXPLAIN EXTENDED show tables").show(truncate = false)
+-----+
| plan
| |
+-----+
|== Parsed Logical Plan ==
ShowTablesCommand

== Analyzed Logical Plan ==
tableName: string, isTemporary: boolean
ShowTablesCommand

== Optimized Logical Plan ==
ShowTablesCommand

== Physical Plan ==
ExecutedCommand
+- ShowTablesCommand|
+-----+
|
```

The following EXPLAIN variants in SQL queries are not supported:

- EXPLAIN FORMATTED
- EXPLAIN LOGICAL

```
scala> sql("EXPLAIN LOGICAL show tables")
org.apache.spark.sql.catalyst.parser.ParseException:
Operation not allowed: EXPLAIN LOGICAL(line 1, pos 0)

== SQL ==
EXPLAIN LOGICAL show tables
^ ^
...
```

The output schema of a `ExplainCommand` is...FIXME

Creating ExplainCommand Instance

`ExplainCommand` takes the following when created:

- [LogicalPlan](#)
- `extended` flag whether to include extended details in the output when `ExplainCommand` is [executed](#) (disabled by default)
- `codegen` flag whether to include codegen details in the output when `ExplainCommand` is [executed](#) (disabled by default)
- `cost` flag whether to include code in the output when `ExplainCommand` is [executed](#) (disabled by default)

`ExplainCommand` initializes [output](#) attribute.

Note	<code>ExplainCommand</code> is created when...FIXME
------	---

Executing Logical Command (Computing Text Representation of QueryExecution) — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note	<code>run</code> is part of RunnableCommand Contract to execute (run) a logical command.
------	--

`run` computes [QueryExecution](#) and returns its text representation in a single `Row`.

Internally, `run` creates a `IncrementalExecution` for a streaming dataset directly or requests `SessionState` to [execute the](#) `LogicalPlan`.

Note	Streaming Dataset is part of Spark Structured Streaming.
------	---

`run` then requests [QueryExecution](#) to build the output text representation, i.e. [codegened](#), [extended](#) (with logical and physical plans), [with stats](#), or [simple](#).

In the end, `run` [creates](#) a `Row` with the text representation.

ExternalRDD

`ExternalRDD` is a [leaf logical operator](#) that is a logical representation of (the data from) an RDD in a logical query plan.

`ExternalRDD` is [created](#) when:

- `SparkSession` is requested to create a [DataFrame](#) from RDD of product types (e.g. Scala case classes, tuples) or [Dataset](#) from RDD of a given type
- `ExternalRDD` is requested to [create a new instance](#)

```
val pairsRDD = sc.parallelize((0, "zero") :: (1, "one") :: (2, "two") :: Nil)

// A tuple of Int and String is a product type
scala> :type pairsRDD
org.apache.spark.rdd.RDD[(Int, String)]

val pairsDF = spark.createDataFrame(pairsRDD)

// ExternalRDD represents the pairsRDD in the query plan
val logicalPlan = pairsDF.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 SerializeFromObject [assertnonnull(assertnonnull(input[0, scala.Tuple2, true]))._1
AS _1#10, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fro
mString, assertnonnull(assertnonnull(input[0, scala.Tuple2, true]))._2, true, false) AS
 _2#11]
01 +- ExternalRDD [obj#9]
```

`ExternalRDD` is a [MultiInstanceRelation](#) and a [ObjectProducer](#).

Note

`ExternalRDD` is resolved to [ExternalRDDScanExec](#) when [BasicOperators](#) execution planning strategy is [executed](#).

newInstance Method

```
newInstance(): LogicalRDD.this.type
```

Note

`newInstance` is part of [MultiInstanceRelation Contract](#) to...FIXME.

`newInstance` ...FIXME

Computing Statistics — `computeStats` Method

```
computeStats(): Statistics
```

Note

`computeStats` is part of [LeafNode Contract](#) to compute statistics for [cost-based optimizer](#).

```
computeStats ...FIXME
```

Creating ExternalRDD Instance

`ExternalRDD` takes the following when created:

- Output schema [attribute](#)
- `RDD` of `T`
- [SparkSession](#)

Creating ExternalRDD — `apply` Factory Method

```
apply[T: Encoder](rdd: RDD[T], session: SparkSession): LogicalPlan
```

```
apply ...FIXME
```

Note

`apply` is used when `SparkSession` is requested to create a [DataFrame](#) from [RDD of product types](#) (e.g. Scala case classes, tuples) or [Dataset](#) from [RDD of a given type](#).

Filter Unary Logical Operator

`Filter` is a [unary logical operator](#) that takes the following when created:

- Condition [expression](#)
- Child [logical operator](#)

`Filter` is [created](#) when...FIXME

Generate Unary Logical Operator for Lateral Views

`Generate` is a [unary logical operator](#) that is [created](#) to represent the following (after a logical plan is [analyzed](#)):

- `Generator` or `GeneratorOuter` expressions (by [ExtractGenerator](#) logical evaluation rule)
- SQL's [LATERAL VIEW](#) clause (in `SELECT` or `FROM` clauses)

`resolved` flag is...FIXME

Note	<code>resolved</code> is part of LogicalPlan Contract to...FIXME.
------	---

`producedAttributes` ...FIXME

The [output schema](#) of a `Generate` is...FIXME

Note	<code>Generate</code> logical operator is resolved to GenerateExec unary physical operator in BasicOperators execution planning strategy.
------	---

	Use <code>generate</code> operator from Catalyst DSL to create a <code>Generate</code> logical operator, e.g. for testing or Spark SQL internals exploration.
--	---

Tip	<pre>import org.apache.spark.sql.catalyst.plans.logical._ import org.apache.spark.sql.types._ val lr = LocalRelation('key.int, 'values.array(StringType)) // JsonTuple generator import org.apache.spark.sql.catalyst.expressions.JsonTuple import org.apache.spark.sql.catalyst.dsl.expressions._ import org.apache.spark.sql.catalyst.expressions.Expression val children: Seq[Expression] = Seq("e") val json_tuple = JsonTuple(children) import org.apache.spark.sql.catalyst.dsl.plans._ // <-- gives generate val plan = lr.generate(generator = json_tuple, join = true, outer = true, alias = Some("alias"), outputNames = Seq.empty) scala> println(plan.numberedTreeString) 00 'Generate json_tuple(e), true, true, alias 01 +- LocalRelation <empty>, [key#0, values#1]</pre>
-----	---

Creating Generate Instance

`Generate` takes the following when created:

- [Generator expression](#)
- [join flag...FIXME](#)
- [outer flag...FIXME](#)
- Optional qualifier
- Output attributes
- Child [logical plan](#)

Generate initializes the [internal registries and counters](#).

GroupingSets Unary Logical Operator

`GroupingSets` is a [unary logical operator](#) that represents SQL's [GROUPING SETS](#) variant of `GROUP BY` clause.

```
val q = sql("""
    SELECT customer, year, SUM(sales)
    FROM VALUES ("abc", 2017, 30) AS t1 (customer, year, sales)
    GROUP BY customer, year
    GROUPING SETS ((customer), (year))
""")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'GroupingSets [ArrayBuffer('customer), ArrayBuffer('year)], ['customer, 'year], ['customer, 'year, unresolvedalias('SUM('sales), None)]
01 +- 'SubqueryAlias t1
02     +- 'UnresolvedInlineTable [customer, year, sales], [List(abc, 2017, 30)]
```

`GroupingSets` operator is resolved to an `Aggregate` logical operator at [analysis phase](#).

```
scala> println(q.queryExecution.analyzed.numberedTreeString)
00 Aggregate [customer#8, year#9, spark_grouping_id#5], [customer#8, year#9, sum(cast(sales#2 as bigint)) AS sum(sales)#4L]
01 +- Expand [List(customer#0, year#1, sales#2, customer#6, null, 1), List(customer#0, year#1, sales#2, null, year#7, 2)], [customer#0, year#1, sales#2, customer#8, year#9, spark_grouping_id#5]
02     +- Project [customer#0, year#1, sales#2, customer#0 AS customer#6, year#1 AS year#7]
03         +- SubqueryAlias t1
04             +- LocalRelation [customer#0, year#1, sales#2]
```

Note	<code>GroupingSets</code> can only be created using SQL.
------	--

Note	<code>GroupingSets</code> is not supported on Structured Streaming's streaming Datasets .
------	---

`GroupingSets` is never resolved (as it can only be converted to an `Aggregate` logical operator).

The [output schema](#) of a `GroupingSets` are exactly the attributes of [aggregate named expressions](#).

Analysis Phase

`GroupingSets` operator is resolved at [analysis phase](#) in the following logical evaluation rules:

- [ResolveAliases](#) for unresolved aliases in aggregate named expressions
- [ResolveGroupingAnalytics](#)

`GroupingSets` operator is resolved to an [Aggregate](#) with [Expand](#) logical operators.

```
val spark: SparkSession = ...
// using q from the example above
val plan = q.queryExecution.logical

scala> println(plan.numberedTreeString)
00 'GroupingSets [ArrayBuffer('customer), ArrayBuffer('year)], ['customer, 'year], ['customer, 'year, unresolvedalias('SUM('sales), None)]
01 +- 'SubqueryAlias t1
02   +- 'UnresolvedInlineTable [customer, year, sales], [List(abc, 2017, 30)]

// Note unresolvedalias for SUM expression
// Note UnresolvedInlineTable and SubqueryAlias

// FIXME Show the evaluation rules to get rid of the unresolvable parts
```

Creating GroupingSets Instance

`GroupingSets` takes the following when created:

- [Expressions](#) from `GROUPING SETS` clause
- Grouping [expressions](#) from `GROUP BY` clause
- Child [logical plan](#)
- Aggregate [named expressions](#)

Hint Logical Operator

Caution	FIXME
---------	-------

HiveTableRelation Leaf Logical Operator — Representing Hive Tables in Logical Plan

`HiveTableRelation` is a [leaf logical operator](#) that represents a Hive table in a [logical query plan](#).

`HiveTableRelation` is [created](#) exclusively when `FindDataSourceTable` logical evaluation rule is requested to [resolve UnresolvedCatalogRelations in a logical plan](#) (for [Hive tables](#)).

```

val tableName = "h1"

// Make the example reproducible
val db = spark.catalog.currentDatabase
import spark.sharedState.{externalCatalog => extCatalog}
extCatalog.dropTable(
  db, table = tableName, ignoreIfNotExists = true, purge = true)

// sql("CREATE TABLE h1 (id LONG) USING hive")
import org.apache.spark.sql.types.StructType
spark.catalog.createTable(
  tableName,
  source = "hive",
  schema = new StructType().add($"id".long),
  options = Map.empty[String, String])

val h1meta = extCatalog.getTable(db, tableName)
scala> println(h1meta.provider.get)
hive

// Looks like we've got the testing space ready for the experiment
val h1 = spark.table(tableName)

import org.apache.spark.sql.catalyst.dsl.plans._
val plan = table(tableName).insertInto("t2", overwrite = true)
scala> println(plan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- 'UnresolvedRelation `h1`

// ResolveRelations logical rule first to resolve UnresolvedRelations
import spark.sessionState.analyzer.ResolveRelations
val rrPlan = ResolveRelations(plan)
scala> println(rrPlan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- 'SubqueryAlias h1
02   +- 'UnresolvedCatalogRelation `default`.`h1`, org.apache.hadoop.hive.serde2.lazy.
LazySimpleSerDe

// FindDataSourceTable logical rule next to resolve UnresolvedCatalogRelations
import org.apache.spark.sql.execution.datasources.FindDataSourceTable
val findTablesRule = new FindDataSourceTable(spark)
val planWithTables = findTablesRule(rrPlan)

// At long last...
// Note HiveTableRelation in the logical plan
scala> println(planWithTables.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- SubqueryAlias h1
02   +- HiveTableRelation `default`.`h1`, org.apache.hadoop.hive.serde2.lazy.LazySimp
leSerDe, [id#13L]

```

`HiveTableRelation` is **partitioned** when it has at least one `partition`.

The `metadata` of a `HiveTableRelation` (in a catalog) has to meet the requirements:

- The `database` is defined
- The `partition schema` is of the same type as `partitionCols`
- The `data schema` is of the same type as `dataCols`

`HiveTableRelation` has the output attributes made up of `data` followed by `partition` columns.

Note	<p><code>HiveTableRelation</code> is removed from a logical plan when <code>HiveAnalysis</code> logical rule is executed (and transforms a <code>InsertIntoTable</code> with <code>HiveTableRelation</code> to an <code>InsertIntoHiveTable</code>).</p> <p><code>HiveTableRelation</code> is when <code>RelationConversions</code> rule is executed (and converts <code>HiveTableRelations</code> to <code>LogicalRelations</code>).</p> <p><code>HiveTableRelation</code> is resolved to <code>HiveTableScanExec</code> physical operator when <code>HiveTableScans</code> strategy is executed.</p>
-------------	---

Computing Statistics — `computeStats` Method

```
computeStats(): Statistics
```

Note	<p><code>computeStats</code> is part of <code>LeafNode Contract</code> to compute statistics for cost-based optimizer.</p>
-------------	--

`computeStats` takes the `table statistics` from the `table metadata` if defined and converts them to `Spark statistics` (with `output columns`).

If the table statistics are not available, `computeStats` reports an `IllegalStateException`.

```
table stats must be specified.
```

Creating HiveTableRelation Instance

`HiveTableRelation` takes the following when created:

- `Table metadata`
- `Columns` (as a collection of `AttributeReferences`)
- `Partitions` (as a collection of `AttributeReferences`)

InMemoryRelation Leaf Logical Operator — Cached Representation of Dataset

`InMemoryRelation` is a [leaf logical operator](#) that represents a cached `dataset` by the [physical query plan](#).

`InMemoryRelation` takes the following to be created:

- Output schema attributes
- `CachedRDDBuilder`
- [Statistics](#) of the `child` query plan
- Output orderings (`Seq[SortOrder]`)

Note

`InMemoryRelation` is usually created using [apply](#) factory methods.

`InMemoryRelation` is [created](#) when:

- `Dataset.persist` operator is used (that in turn requests `CacheManager` to [cache a structured query](#))
- `CatalogImpl` is requested to [cache](#) or [refresh](#) a table or view in-memory
- `InsertIntoDataSourceCommand` logical command is [executed](#) (and in turn requests `CacheManager` to [recacheByPlan](#))
- `CatalogImpl` is requested to [refreshByPath](#) (and in turn requests `CacheManager` to [recacheByPath](#))
- `QueryExecution` is requested for a [cached logical query plan](#) (and in turn requests `CacheManager` to [replace logical query segments with cached query plans](#))

```

// Cache sample table range5 using pure SQL
// That registers range5 to contain the output of range(5) function
spark.sql("CACHE TABLE range5 AS SELECT * FROM range(5)")
val q1 = spark.sql("SELECT * FROM range5")
scala> q1.explain
== Physical Plan ==
InMemoryTableScan [id#0L]
+- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1
replicas), `range5`
    +- *Range (0, 5, step=1, splits=8)

// you could also use optimizedPlan to see InMemoryRelation
scala> println(q1.queryExecution.optimizedPlan.numberedTreeString)
00 InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 r
eplicas), `range5`
01     +- *Range (0, 5, step=1, splits=8)

// Use Dataset's cache
val q2 = spark.range(10).groupBy('id % 5).count.cache
scala> println(q2.queryExecution.optimizedPlan.numberedTreeString)
00 InMemoryRelation [(id % 5)#84L, count#83L], true, 10000, StorageLevel(disk, memory,
deserialized, 1 replicas)
01     +- *HashAggregate(keys=[(id#77L % 5)#88L], functions=[count(1)], output=[(id % 5
)#84L, count#83L])
02         +- Exchange hashpartitioning((id#77L % 5)#88L, 200)
03             +- *HashAggregate(keys=[(id#77L % 5) AS (id#77L % 5)#88L], functions=[part
ial_count(1)], output=[(id#77L % 5)#88L, count#90L])
04                 +- *Range (0, 10, step=1, splits=8)

```

`InMemoryRelation` is a [MultInstanceRelation](#) so a new instance will be created to appear

multiple times in a physical query plan.

```
// Cache a Dataset
val q = spark.range(10).cache

// Make sure that q Dataset is cached
val cache = spark.sharedState.cacheManager
scala> cache.lookupCachedData(q.queryExecution.logical).isDefined
res0: Boolean = true

scala> q.explain
== Physical Plan ==
InMemoryTableScan [id#122L]
+- InMemoryRelation [id#122L], true, 10000, StorageLevel(disk, memory, deserialized
, 1 replicas)
    +- *Range (0, 10, step=1, splits=8)

val qCrossJoined = q.crossJoin(q)
scala> println(qCrossJoined.queryExecution.optimizedPlan.numberedTreeString)
00 Join Cross
01 :- InMemoryRelation [id#122L], true, 10000, StorageLevel(disk, memory, deserialized
, 1 replicas)
02 :   +- *Range (0, 10, step=1, splits=8)
03 +- InMemoryRelation [id#170L], true, 10000, StorageLevel(disk, memory, deserialized
, 1 replicas)
04       +- *Range (0, 10, step=1, splits=8)

// Use sameResult for comparison
// since the plans use different output attributes
// and have to be canonicalized internally
import org.apache.spark.sql.execution.columnar.InMemoryRelation
val optimizedPlan = qCrossJoined.queryExecution.optimizedPlan
scala> optimizedPlan.children(0).sameResult(optimizedPlan.children(1))
res1: Boolean = true
```

The [simple text representation](#) of a `InMemoryRelation` (aka `simpleString`) is [InMemoryRelation \[output\], \[storageLevel\]](#) (that uses the `output` and the `CachedRDDBuilder`).

```
val q = spark.range(1).cache
val logicalPlan = q.queryExecution.withCachedData
scala> println(logicalPlan.simpleString)
InMemoryRelation [id#40L], StorageLevel(disk, memory, deserialized, 1 replicas)
```

`InMemoryRelation` is resolved to [InMemoryTableScanExec](#) leaf physical operator when [InMemoryScans](#) execution planning strategy is executed.

Table 1. InMemoryRelation's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
partitionStatistics	<p><code>PartitionStatistics</code> for the <code>output</code> schema</p> <p>Used exclusively when <code>InMemoryTableScanExec</code> is <code>created</code> (and initializes <code>stats</code> internal property).</p>

Computing Statistics — `computeStats` Method

```
computeStats(): Statistics
```

Note	<code>computeStats</code> is part of LeafNode Contract to compute statistics for cost-based optimizer .
------	---

```
computeStats ...FIXME
```

`withOutput` Method

```
withOutput(newOutput: Seq[Attribute]): InMemoryRelation
```

```
withOutput ...FIXME
```

Note	<code>withOutput</code> is used exclusively when <code>CacheManager</code> is requested to replace logical query segments with cached query plans .
------	---

`newInstance` Method

```
newInstance(): this.type
```

Note	<code>newInstance</code> is part of MultilnstanceRelation Contract to...FIXME.
------	--

```
newInstance ...FIXME
```

`cachedColumnBuffers` Method

```
cachedColumnBuffers: RDD[CachedBatch]
```

```
cachedColumnBuffers ...FIXME
```

Note	<code>cachedColumnBuffers</code> is used when...FIXME
------	---

PartitionStatistics

```
PartitionStatistics(tableSchema: Seq[Attribute])
```

Note	<code>PartitionStatistics</code> is a <code>private[columnar]</code> class.
------	---

`PartitionStatistics` ...FIXME

Note	<code>PartitionStatistics</code> is used exclusively when <code>InMemoryRelation</code> is created (and initializes <code>partitionStatistics</code>).
------	---

Creating InMemoryRelation Instance — apply Factory Methods

```
apply(  
  useCompression: Boolean,  
  batchSize: Int,  
  storageLevel: StorageLevel,  
  child: SparkPlan,  
  tableName: Option[String],  
  logicalPlan: LogicalPlan): InMemoryRelation  
apply(  
  cacheBuilder: CachedRDDBuilder,  
  logicalPlan: LogicalPlan): InMemoryRelation
```

`apply` creates an `InMemoryRelation` logical operator.

Note	<code>apply</code> is used when <code>CacheManager</code> is requested to <code>cache</code> and <code>re-cache</code> a structured query.
------	--

InsertIntoDataSourceCommand Logical Command

`InsertIntoDataSourceCommand` is a `RunnableCommand` that inserts or overwrites data in an `InsertableRelation` (per `overwrite` flag).

`InsertIntoDataSourceCommand` is created exclusively when `DataSourceAnalysis` logical resolution is executed (and resolves an `InsertIntoTable` unary logical operator with a `LogicalRelation` on an `InsertableRelation`).

```
sql("DROP TABLE IF EXISTS t2")
sql("CREATE TABLE t2(id long)")

val query = "SELECT * FROM RANGE(1)"
// Using INSERT INTO SQL statement so we can access QueryExecution
// DataFrameWriter.insertInto returns no value
val q = sql("INSERT INTO TABLE t2 " + query)
val logicalPlan = q.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, false, false
01 +- 'Project [*]
02   +- 'UnresolvedTableValuedFunction RANGE, [1]

val analyzedPlan = q.queryExecution.analyzed
scala> println(analyzedPlan.numberedTreeString)
00 InsertIntoHiveTable `default`.`t2`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSe
rDe, false, false, [id#6L]
01 +- Project [id#6L]
02   +- Range (0, 1, step=1, splits=None)
```

`InsertIntoDataSourceCommand` returns the logical query plan when requested for the inner nodes (that should be shown as an inner nested tree of this node).

```
val query = "SELECT * FROM RANGE(1)"
val sqlText = "INSERT INTO TABLE t2 " + query
val plan = spark.sessionState.sqlParser.parsePlan(sqlText)
scala> println(plan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, false, false
01 +- 'Project [*]
02   +- 'UnresolvedTableValuedFunction RANGE, [1]
```

Executing Logical Command (Inserting or Overwriting Data in `InsertableRelation`) — `run` Method

```
run(session: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (`run`) a logical command.

`run` takes the [InsertableRelation](#) (that is the relation of the [LogicalRelation](#)).

`run` then creates a [DataFrame](#) for the [logical query plan](#) and the input `SparkSession`.

`run` requests the `DataFrame` for the [QueryExecution](#) that in turn is requested for the [RDD](#) (of the structured query). `run` requests the [LogicalRelation](#) for the [output schema](#).

With the [RDD](#) and the output schema, `run` creates another [DataFrame](#) that is the `RDD[InternalRow]` with the schema applied.

`run` requests the `InsertableRelation` to [insert or overwrite data](#).

In the end, since the data in the `InsertableRelation` has changed, `run` requests the `CacheManager` to [recacheByPlan](#) with the [LogicalRelation](#).

Note

`run` requests the `SparkSession` for [SharedState](#) that is in turn requested for the [CacheManager](#).

Creating InsertIntoDataSourceCommand Instance

`InsertIntoDataSourceCommand` takes the following when created:

- [LogicalRelation](#) leaf logical operator
- [Logical query plan](#)
- `overwrite` flag

InsertIntoDataSourceDirCommand Logical Command

`InsertIntoDataSourceDirCommand` is a logical command that [FIXME](#).

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` ...[FIXME](#)

InsertIntoDir Unary Logical Operator

InsertIntoDir is...FIXME

InsertIntoHadoopFsRelationCommand Logical Command

`InsertIntoHadoopFsRelationCommand` is a concrete [DataWritingCommand](#) that inserts the result of executing a [query](#) to an [output path](#) in the given [FileFormat](#) (and [other properties](#)).

`InsertIntoHadoopFsRelationCommand` is [created](#) when:

- `DataSource` is requested to [planForWritingFileFormat](#)
- [DataSourceAnalysis](#) post-hoc logical resolution rule is executed (and resolves a `InsertIntoTable` logical operator with a [HadoopFsRelation](#))

`InsertIntoHadoopFsRelationCommand` uses **partitionOverwriteMode** option that overrides `spark.sql.sources.partitionOverwriteMode` property for dynamic partition inserts.

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession, child: SparkPlan): Seq[Row]
```

Note	<code>run</code> is part of DataWritingCommand Contract to execute (run) a logical command to write query data
------	--

`run` uses the `spark.sql.hive.manageFilesourcePartitions` configuration property to...FIXME

Caution	FIXME When is the <code>catalogTable</code> defined?
---------	--

Caution	FIXME When is <code>tracksPartitionsInCatalog</code> of <code>CatalogTable</code> enabled?
---------	--

`run` gets the `partitionOverwriteMode` option...FIXME

`run` creates a `FileCommitProtocol` for the `spark.sql.sources.commitProtocolClass` (default: `SQLHadoopMapReduceCommitProtocol`) and the `outputPath`, the `dynamicPartitionOverwrite`, and random jobId.

For insertion, `run` simply requests the `FileFormatWriter` object to `write` and then...FIXME (does some table-specific "tasks").

Otherwise (for non-insertion case), `run` simply prints out the following INFO message to the logs and finishes.

```
Skipping insertion into a relation that already exists.
```

`run` uses `SchemaUtils` to make sure that there are no duplicates in the `outputColumnNames`.

Creating InsertIntoHadoopFsRelationCommand Instance

`InsertIntoHadoopFsRelationCommand` takes the following when created:

- Output Hadoop's [Path](#)
- Static table partitions (`Map[String, String]`)
- `ifPartitionNotExists` flag
- Partition columns (`Seq[Attribute]`)
- [BucketSpec](#)
- [FileFormat](#)
- Options (`Map[String, String]`)
- [Logical plan](#)
- `SaveMode`
- [CatalogTable](#)
- `FileInfo`
- Output column names

Note

`staticPartitions` may hold zero or more partitions as follows:

- Always empty when `created` when `DataSource` is requested to `planForWritingFileFormat`
- Possibly with partitions when `created` when `DataSourceAnalysis` posthoc logical resolution rule is [applied](#) to an [InsertIntoTable](#) logical operator over a [HadoopFsRelation](#) relation

With that, `staticPartitions` are simply the partitions of an [InsertIntoTable](#) logical operator.

InsertIntoHiveDirCommand Logical Command

InsertIntoHiveDirCommand is...FIXME

Executing Logical Command — run Method

```
run(session: SparkSession): Seq[Row]
```

Note

run is part of [RunnableCommand Contract](#) to execute (run) a logical command.

run ...FIXME

InsertIntoHiveTable Logical Command

`InsertIntoHiveTable` is...FIXME

Executing Logical Command — `run` Method

```
run(session: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` ...FIXME

processInsert Internal Method

```
processInsert(  
    sparkSession: SparkSession,  
    externalCatalog: ExternalCatalog,  
    hadoopConf: Configuration,  
    tableDesc: TableDesc,  
    tmpLocation: Path,  
    child: SparkPlan): Unit
```

`processInsert` ...FIXME

Note

`processInsert` is used exclusively when `InsertIntoHiveTable` logical command is [executed](#).

InsertIntoTable Unary Logical Operator

`InsertIntoTable` is a [unary logical operator](#) that represents the following:

- `INSERT INTO` and `INSERT OVERWRITE TABLE` SQL statements
- `DataFrameWriter` is requested to insert the rows of a DataFrame into a table

```
// make sure that the tables are available in a catalog
sql("CREATE TABLE IF NOT EXISTS t1(id long)")
sql("CREATE TABLE IF NOT EXISTS t2(id long)")

val q = sql("INSERT INTO TABLE t2 SELECT * from t1 LIMIT 100")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, false, false
01 +- 'GlobalLimit 100
02     +- 'LocalLimit 100
03         +- 'Project [*]
04             +- 'UnresolvedRelation `t1`

// Dataset API's version of "INSERT OVERWRITE TABLE" in SQL
spark.range(10).write.mode("overwrite").insertInto("t2")
```

INSERT INTO partitioned_table

```
spark.range(10)
  .withColumn("p1", 'id % 2')
  .write
  .mode("overwrite")
  .partitionBy("p1")
  .saveAsTable("partitioned_table")

val insertIntoQ = sql("INSERT INTO TABLE partitioned_table PARTITION (p1 = 4) VALUES 4
1, 42")
scala> println(insertIntoQ.queryExecution.logical.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `partitioned_table`, Map(p1 -> Some(4)), false
, false
01 +- 'UnresolvedInlineTable [col1], [List(41), List(42)]
```

INSERT OVERWRITE TABLE partitioned_table

```

spark.range(10)
  .withColumn("p1", 'id % 2)
  .write
  .mode("overwrite")
  .partitionBy("p1")
  .saveAsTable("partitioned_table")

val insertOverwriteQ = sql("INSERT OVERWRITE TABLE partitioned_table PARTITION (p1 = 4
) VALUES 40")
scala> println(insertOverwriteQ.queryExecution.logical.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `partitioned_table`, Map(p1 -> Some(4)), true,
false
01 +- 'UnresolvedInlineTable [col1], [List(40)]

```

`InsertIntoTable` is created with `partition keys` that correspond to the `partitionspec` part of the following SQL statements:

- `INSERT INTO TABLE` (with the `overwrite` and `ifPartitionNotExists` flags off)
- `INSERT OVERWRITE TABLE` (with the `overwrite` and `ifPartitionNotExists` flags off)

`InsertIntoTable` has no `partition keys` when created as follows:

- `insertInto` operator from the [Catalyst DSL](#)
- `DataFrameWriter.insertInto` operator

`InsertIntoTable` can never be [resolved](#) (i.e. `InsertIntoTable` should not be part of a logical plan after analysis and is supposed to be [converted to logical commands](#) at analysis phase).

Table 1. `InsertIntoTable`'s Logical Resolutions (Conversions)

Logical Command	Description
<code>InsertIntoHiveTable</code>	When HiveAnalysis resolution rule transforms <code>InsertIntoTable</code> with a HiveTableRelation
<code>InsertIntoDataSourceCommand</code>	When DataSourceAnalysis posthoc logical resolution resolves an <code>InsertIntoTable</code> with a LogicalRelation over an InsertableRelation (with no partitions defined)
<code>InsertIntoHadoopFsRelationCommand</code>	When DataSourceAnalysis posthoc logical resolution transforms <code>InsertIntoTable</code> with a LogicalRelation over a HadoopFsRelation

Caution	<p><small>FIXME What's the difference between <code>HiveAnalysis</code> that converts <code>InsertIntoTable(r: HiveTableRelation...)</code> to <code>InsertIntoHiveTable</code> and <code>RelationConversions</code> that converts <code>InsertIntoTable(r: HiveTableRelation,...)</code> to <code>InsertIntoTable</code> (with <code>LogicalRelation</code>)?</small></p>
---------	---

Note	<p><code>InsertIntoTable</code> (with <code>UnresolvedRelation</code> leaf logical operator) is created when:</p>
------	---

- `INSERT INTO` or `INSERT OVERWRITE TABLE` SQL statements are executed (as a [single insert](#) or a [multi-insert](#) query)
- `DataFrameWriter` is requested to [insert a DataFrame into a table](#)
- `RelationConversions` logical evaluation rule is [executed](#) (and transforms `InsertIntoTable` operators)
- `CreateHiveTableAsSelectCommand` logical command is [executed](#)

`InsertIntoTable` has an empty [output schema](#).

Tip	<p>Use <code>insertInto</code> operator from the Catalyst DSL to create an <code>InsertIntoTable</code> logical operator, e.g. for testing or Spark SQL internals exploration.</p> <pre>import org.apache.spark.sql.catalyst.dsl.plans._ val plan = table("a").insertInto(tableName = "t1", overwrite = true) scala> println(plan.numberedTreeString) 00 'InsertIntoTable 'UnresolvedRelation `t1`, true, false 01 +- 'UnresolvedRelation `a` import org.apache.spark.sql.catalyst.plans.logical.InsertIntoTable val op = plan.p(0).asInstanceOf[InsertIntoTable] scala> :type op org.apache.spark.sql.catalyst.plans.logical.InsertIntoTable</pre>
-----	--

Creating `InsertIntoTable` Instance

`InsertIntoTable` takes the following when created:

- [Logical plan](#) for the table to insert into
- Partition keys (with optional partition values for [dynamic partition insert](#))
- [Logical plan](#) representing the data to be written
- `overwrite` flag that indicates whether to overwrite an existing table or partitions (`true`) or not (`false`)
- `ifPartitionNotExists` flag

Inserting Into View Not Allowed

Inserting into a view is not allowed, i.e. a query plan with an `InsertIntoTable` operator with a `UnresolvedRelation` leaf operator that is resolved to a `View` unary operator fails at analysis (when `ResolveRelations` logical resolution is executed).

```
Inserting into a view is not allowed. View: [name].
```

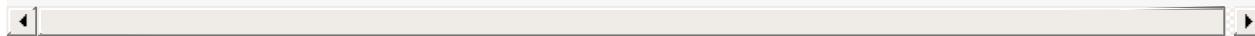
```
// Create a view
val viewName = "demo_view"
sql(s"DROP VIEW IF EXISTS $viewName")
assert(spark.catalog.tableExists(viewName) == false)
sql(s"CREATE VIEW $viewName COMMENT 'demo view' AS SELECT 1,2,3")
assert(spark.catalog.tableExists(viewName))

// The following should fail with an AnalysisException
scala> spark.range(0).write.insertInto(viewName)
org.apache.spark.sql.AnalysisException: Inserting into a view is not allowed. View: `d
efault`.`demo_view`.;
  at org.apache.spark.sql.catalyst.analysis.package$AnalysisErrorAt.failAnalysis(package.scala:42)
  at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveRelations$$anonfun$apply$8.
applyOrElse(Analyzer.scala:644)
  at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveRelations$$anonfun$apply$8.
applyOrElse(Analyzer.scala:640)
  at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$transformUp$1.apply(TreeNode
.scala:289)
  at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$transformUp$1.apply(TreeNode
.scala:289)
  at org.apache.spark.sql.catalyst.trees.CurrentOrigin$.withOrigin(TreeNode.scala:70)
  at org.apache.spark.sql.catalyst.trees.TreeNode.transformUp(TreeNode.scala:288)
  at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveRelations$.apply(Analyzer.
scala:640)
  at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveRelations$.apply(Analyzer.
scala:586)
  at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1$$anonfun$appl
y$1.apply(RuleExecutor.scala:87)
  at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1$$anonfun$appl
y$1.apply(RuleExecutor.scala:84)
  at scala.collection.LinearSeqOptimized$class.foldLeft(LinearSeqOptimized.scala:124)
  at scala.collection.immutable.List.foldLeft(List.scala:84)
  at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1.apply(RuleExe
cutor.scala:84)
  at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1.apply(RuleExe
cutor.scala:76)
  at scala.collection.immutable.List.foreach(List.scala:381)
  at org.apache.spark.sql.catalyst.rules.RuleExecutor.execute(RuleExecutor.scala:76)
  at org.apache.spark.sql.catalyst.analysis.Analyzer.org$apache$spark$sql$catalyst$ana
lysis$Analyzer$$executeSameContext(Analyzer.scala:124)
  at org.apache.spark.sql.catalyst.analysis.Analyzer.execute(Analyzer.scala:118)
```

```

    at org.apache.spark.sql.catalyst.analysis.Analyzer.executeAndCheck(Analyzer.scala:103
)
    at org.apache.spark.sql.execution.QueryExecution.analyzed$lzycompute(QueryExecution.
scala:57)
    at org.apache.spark.sql.execution.QueryExecution.analyzed(QueryExecution.scala:55)
    at org.apache.spark.sql.execution.QueryExecution.assertAnalyzed(QueryExecution.scala:
47)
    at org.apache.spark.sql.execution.QueryExecution.withCachedData$lzycompute(QueryExec
ution.scala:61)
    at org.apache.spark.sql.execution.QueryExecution.withCachedData(QueryExecution.scala:
60)
    at org.apache.spark.sql.execution.QueryExecution.optimizedPlan$lzycompute(QueryExecu
tion.scala:66)
    at org.apache.spark.sql.execution.QueryExecution.optimizedPlan(QueryExecution.scala:
66)
    at org.apache.spark.sql.execution.QueryExecution.sparkPlan$lzycompute(QueryExecution
.scala:72)
    at org.apache.spark.sql.execution.QueryExecution.sparkPlan(QueryExecution.scala:68)
    at org.apache.spark.sql.execution.QueryExecution.executedPlan$lzycompute(QueryExecut
ion.scala:77)
    at org.apache.spark.sql.execution.QueryExecution.executedPlan(QueryExecution.scala:77
)
    at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.sca
la:75)
    at org.apache.spark.sql.DataFrameWriter.runCommand(DataFrameWriter.scala:654)
    at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:322)
    at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:308)
    ... 49 elided

```



Inserting Into RDD-Based Table Not Allowed

Inserting into an RDD-based table is not allowed, i.e. a query plan with an `InsertIntoTable` operator with one of the following logical operators (as the [logical plan representing the table](#)) fails at analysis (when `PreWriteCheck` extended logical check is executed):

- Logical operator is not a [leaf node](#)
- [Range](#) leaf operator
- [OneRowRelation](#) leaf operator
- [LocalRelation](#) leaf operator

```

// Create a temporary view
val data = spark.range(1)
data.createOrReplaceTempView("demo")

scala> spark.range(0).write.insertInto("demo")
org.apache.spark.sql.AnalysisException: Inserting into an RDD-based table is not allow

```

```

ed.;

'InsertIntoTable Range (0, 1, step=1, splits=Some(8)), false, false
+- Range (0, 0, step=1, splits=Some(8))

    at org.apache.spark.sql.execution.datasources.PreWriteCheck$.failAnalysis(rules.scala:442)
    at org.apache.spark.sql.execution.datasources.PreWriteCheck$$anonfun$apply$14.apply(rules.scala:473)
    at org.apache.spark.sql.execution.datasources.PreWriteCheck$$anonfun$apply$14.apply(rules.scala:445)
    at org.apache.spark.sql.catalyst.trees.TreeNode.foreach(TreeNode.scala:117)
    at org.apache.spark.sql.execution.datasources.PreWriteCheck$.apply(rules.scala:446)
    at org.apache.spark.sql.execution.datasources.PreWriteCheck$.apply(rules.scala:447)
    at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$2.apply(CheckAnalysis.scala:349)
    at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$2.apply(CheckAnalysis.scala:349)
    at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:59)
    at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
    at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$class.checkAnalysis(CheckAnalysis.scala:349)
    at org.apache.spark.sql.catalyst.analysis.Analyzer.checkAnalysis(Analyzer.scala:92)
    at org.apache.spark.sql.catalyst.analysis.Analyzer.executeAndCheck(Analyzer.scala:105)
)
    at org.apache.spark.sql.execution.QueryExecution.analyzed$lzycompute(QueryExecution.scala:57)
    at org.apache.spark.sql.execution.QueryExecution.analyzed(QueryExecution.scala:55)
    at org.apache.spark.sql.execution.QueryExecution.assertAnalyzed(QueryExecution.scala:47)
    at org.apache.spark.sql.execution.QueryExecution.withCachedData$lzycompute(QueryExecution.scala:61)
    at org.apache.spark.sql.execution.QueryExecution.withCachedData(QueryExecution.scala:60)
    at org.apache.spark.sql.execution.QueryExecution.optimizedPlan$lzycompute(QueryExecution.scala:66)
    at org.apache.spark.sql.execution.QueryExecution.optimizedPlan(QueryExecution.scala:66)
    at org.apache.spark.sql.execution.QueryExecution.sparkPlan$lzycompute(QueryExecution.scala:72)
    at org.apache.spark.sql.execution.QueryExecution.sparkPlan(QueryExecution.scala:68)
    at org.apache.spark.sql.execution.QueryExecution.executedPlan$lzycompute(QueryExecution.scala:77)
    at org.apache.spark.sql.execution.QueryExecution.executedPlan(QueryExecution.scala:77)
)
    at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.scala:75)
    at org.apache.spark.sql.DataFrameWriter.runCommand(DataFrameWriter.scala:654)
    at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:322)
    at org.apache.spark.sql.DataFrameWriter.insertInto(DataFrameWriter.scala:308)
    ... 49 elided

```


Intersect

Intersect is...FIXME

Join Logical Operator

`Join` is a [binary logical operator](#), i.e. works with two logical operators. `Join` has a join type and an optional expression condition for the join.

`Join` is [created](#) when...FIXME

Note	CROSS JOIN is just an INNER JOIN with no join condition.
------	--

`Join` has [output schema attributes](#)...FIXME

Creating Join Instance

`Join` takes the following when created:

- [Logical plan](#) of the left side
- [Logical plan](#) of the right side
- [Join type](#)
- Join condition (if available) as a [Catalyst expression](#)

LeafNode — Base Logical Operator with No Child Operators and Optional Statistics

`LeafNode` is the base of [logical operators](#) that have no `child` operators.

`LeafNode` that wants to survive analysis has to define `computeStats` as it throws an `UnsupportedOperationException` by default.

Table 1. LeafNodes (Direct Implementations)

LeafNode	Description
AnalysisBarrier	
DataSourceV2Relation	
ExternalRDD	
HiveTableRelation	
InMemoryRelation	
LocalRelation	
LogicalRDD	
LogicalRelation	
OneRowRelation	
Range	
UnresolvedCatalogRelation	
UnresolvedInlineTable	
UnresolvedRelation	
UnresolvedTableValuedFunction	

Computing Statistics — `computeStats` Method

```
computeStats(): Statistics
```

`computeStats` simply throws an `UnsupportedOperationException`.

Note

Logical operators, e.g. `ExternalRDD`, `LogicalRDD` and `DataSourceV2Relation`, or relations, e.g. `HadoopFsRelation` or `BaseRelation`, use `spark.sql.defaultSizeInBytes` internal property for the default estimated size if the statistics could not be computed.

Note

`computeStats` is used exclusively when `SizeInBytesOnlyStatsPlanVisitor` uses the `default case` to compute the size statistic (in bytes) for a `logical operator`.

LocalRelation Leaf Logical Operator

`LocalRelation` is a [leaf logical operator](#) that represents a scan over local collections (and allow for optimizations so functions like `collect` or `take` can be executed locally on the driver and with no executors).

`LocalRelation` is [created](#) (using `apply`, `fromExternalRows`, and `fromProduct` factory methods) when:

- `ResolveInlineTables` logical resolution rule is [executed](#) (and [converts an `UnresolvedInlineTable`](#))
- `PruneFilters`, `ConvertToLocalRelation`, and `PropagateEmptyRelation`, `OptimizeMetadataOnlyQuery` logical optimization rules are executed (applied to an analyzed logical plan)
- `SparkSession.createDataset`, `SparkSession.emptyDataset`, `SparkSession.createDataFrame` operators are used
- `CatalogImpl` is requested for a `Dataset` from `DefinedByConstructorParams` data
- `Dataset` is requested for the [analyzed logical plan](#) (and executes `Command` logical operators)
- `StatFunctions` is requested to `crossTabulate` and [generate summary statistics of Dataset \(as DataFrame\)](#)

Note	Dataset is local when the analyzed logical plan is exactly an instance of <code>LocalRelation</code> .
------	--

```

val data = Seq(1, 3, 4, 7)
val nums = data.toDF

scala> :type nums
org.apache.spark.sql.DataFrame

val plan = nums.queryExecution.analyzed
scala> println(plan.numberedTreeString)
00 LocalRelation [value#1]

import org.apache.spark.sql.catalyst.plans.logical.LocalRelation
val relation = plan.collect { case r: LocalRelation => r }.head
assert(relation.isInstanceOf[LocalRelation])

val sql = relation.toSQL(inlineTableName = "demo")
assert(sql == "VALUES (1), (3), (4), (7) AS demo(value)")

val stats = relation.computeStats
scala> println(stats)
Statistics(sizeInBytes=48.0 B, hints=null)

```

`LocalRelation` is resolved to [LocalTableScanExec](#) leaf physical operator when [BasicOperators](#) execution planning strategy is executed (i.e. plan a [logical plan](#) to a [physical plan](#)).

```

import org.apache.spark.sql.catalyst.plans.logical.LocalRelation
assert(relation.isInstanceOf[LocalRelation])

scala> :type spark
org.apache.spark.sql.SparkSession

import spark.sessionState.planner.BasicOperators
val localScan = BasicOperators(relation).head

import org.apache.spark.sql.execution.LocalTableScanExec
assert(localScan.isInstanceOf[LocalTableScanExec])

```

When requested for [statistics](#), `LocalRelation` takes the size of the objects in a single row (per the [output schema](#)) and multiplies it by the number of rows (in the [data](#)).

Creating LocalRelation Instance

`LocalRelation` takes the following to be created:

- Output schema [attributes](#)
- Collection of [internal binary rows](#)

- `isStreaming` flag that indicates whether the `data` comes from a streaming source
(default: `false`)

While being created, `LocalRelation` makes sure that the `output attributes` are all `resolved` or throws an `IllegalArgumentException`:

```
Unresolved attributes found when constructing LocalRelation.
```

Creating LocalRelation — `apply` Object Method

```
apply(output: Attribute*): LocalRelation
apply(
    output1: StructField,
    output: StructField*): LocalRelation
```

`apply` ...FIXME

Note	<code>apply</code> is used when...FIXME
------	---

Creating LocalRelation — `fromExternalRows` Object Method

```
fromExternalRows(
    output: Seq[Attribute],
    data: Seq[Row]): LocalRelation
```

`fromExternalRows` ...FIXME

Note	<code>fromExternalRows</code> is used when...FIXME
------	--

Creating LocalRelation — `fromProduct` Object Method

```
fromProduct(
    output: Seq[Attribute],
    data: Seq[Product]): LocalRelation
```

`fromProduct` ...FIXME

Note	<code>fromProduct</code> is used when...FIXME
------	---

Generating SQL Statement— `toSQL` Method

```
toSQL(inlineTableName: String): String
```

`toSQL` generates a SQL statement of the format:

```
VALUES [data] AS [inlineTableName]([names])
```

`toSQL` throws an `AssertionError` for the `data` empty.

Note

`toSQL` does not seem to be used at all.

LogicalRDD — Logical Scan Over RDD

`LogicalRDD` is a [leaf logical operator](#) with [MultiInstanceRelation](#) support for a logical representation of a scan over [RDD of internal binary rows](#).

`LogicalRDD` is [created](#) when:

- `Dataset` is requested to [checkpoint](#)
- `SparkSession` is requested to [create a DataFrame from an RDD of internal binary rows](#)

Note

`LogicalRDD` is resolved to [RDDScanExec](#) when `BasicOperators` execution planning strategy is [executed](#).

newInstance Method

```
newInstance(): LogicalRDD.this.type
```

Note

`newInstance` is part of [MultiInstanceRelation Contract](#) to...FIXME.

newInstance ...FIXME

Computing Statistics — computeStats Method

```
computeStats(): Statistics
```

Note

`computeStats` is part of [LeafNode Contract](#) to compute statistics for [cost-based optimizer](#).

computeStats ...FIXME

Creating LogicalRDD Instance

`LogicalRDD` takes the following when created:

- Output schema attributes
- `RDD` of [internal binary rows](#)
- [Partitioning](#)
- Output ordering (`SortOrder`)

- `isStreaming flag`
- [SparkSession](#)

LogicalRelation Leaf Logical Operator — Representing BaseRelations in Logical Plan

`LogicalRelation` is a [leaf logical operator](#) that represents a [BaseRelation](#) in a logical query plan.

```
val q1 = spark.read.option("header", true).csv("../datasets/people.csv")
scala> println(q1.queryExecution.logical.numberedTreeString)
00 Relation[id#72,name#73,age#74] csv

val q2 = sql("select * from `csv`..../datasets/people.csv`")
scala> println(q2.queryExecution.optimizedPlan.numberedTreeString)
00 Relation[_c0#175,_c1#176,_c2#177] csv
```

`LogicalRelation` is [created](#) when:

- `DataFrameReader` loads data from a data source that supports multiple paths (through `SparkSession.baseRelationToDataFrame`)
- `DataFrameReader` is requested to load data from an external table using `JDBC` (through `SparkSession.baseRelationToDataFrame`)
- `TextInputCSVDataSource` and `TextInputJsonDataSource` are requested to infer schema
- `ResolveSQLOnFile` converts a logical plan
- `FindDataSourceTable` logical evaluation rule is [executed](#)
- `RelationConversions` logical evaluation rule is [executed](#)
- `CreateTempViewUsing` logical command is requested to [run](#)
- Structured Streaming's `FileStreamSource` creates batches of records

<p>Note</p>	<p><code>LogicalRelation</code> can be created using <code>apply</code> factory methods that accept <code>BaseRelation</code> with optional <code>CatalogTable</code>.</p>
--------------------	--

```
apply(relation: BaseRelation): LogicalRelation
apply(relation: BaseRelation, table: CatalogTable): LogicalRelation
```

The [simple text representation](#) of a `LogicalRelation` (aka `simpleString`) is `Relation[output] [relation]` (that uses the `output` and `BaseRelation`).

```
val q = spark.read.text("README.md")
val logicalPlan = q.queryExecution.logical
scala> println(logicalPlan.simpleString)
Relation[value#2] text
```

refresh Method

```
refresh(): Unit
```

Note

`refresh` is part of [LogicalPlan Contract](#) to refresh itself.

`refresh` requests the [FileIndex](#) of a [HadoopFsRelation](#) `relation` to refresh.

Note

`refresh` does the work for [HadoopFsRelation](#) relations only.

Creating LogicalRelation Instance

`LogicalRelation` takes the following when created:

- [BaseRelation](#)
- Output schema [AttributeReferences](#)
- Optional [CatalogTable](#)

OneRowRelation Leaf Logical Operator

OneRowRelation is a leaf logical operator that...FIXME

Pivot Unary Logical Operator

Pivot is a unary logical operator that represents pivot operator.

```
val visits = Seq(
  (0, "Warsaw", 2015),
  (1, "Warsaw", 2016),
  (2, "Boston", 2017)
).toDF("id", "city", "year")

val q = visits
  .groupBy("city")
  .pivot("year", Seq("2015", "2016", "2017"))
  .count()
scala> println(q.queryExecution.logical.numberedTreeString)
00 Pivot [city#8], year#9: int, [2015, 2016, 2017], [count(1) AS count#157L]
01 +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
02   +- LocalRelation [_1#3, _2#4, _3#5]
```

Pivot is created when RelationalGroupedDataset creates a DataFrame for an aggregate operator.

Analysis Phase

Pivot operator is resolved at analysis phase in the following logical evaluation rules:

- ResolveAliases
- ResolvePivot

```
val spark: SparkSession = ...

import spark.sessionState.analyzer.ResolveAliases
// see q in the example above
val plan = q.queryExecution.logical

scala> println(plan.numberedTreeString)
00 Pivot [city#8], year#9: int, [2015, 2016, 2017], [count(1) AS count#24L]
01 +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
02   +- LocalRelation [_1#3, _2#4, _3#5]

// FIXME Find a plan to show the effect of ResolveAliases
val planResolved = ResolveAliases(plan)
```

Pivot operator "disappears" behind (i.e. is converted to) a [Aggregate logical operator](#) (possibly under `Project` operator).

```
import spark.sessionState.analyzer.ResolvePivot
val planAfterResolvePivot = ResolvePivot(plan)
scala> println(planAfterResolvePivot.numberedTreeString)
00 Project [city#8, __pivot_count(1) AS `count` AS `count(1) AS ``count``#62[0] AS 20
15#63L, __pivot_count(1) AS `count` AS `count(1) AS ``count``#62[1] AS 2016#64L, __pi
pivot_count(1) AS `count` AS `count(1) AS ``count``#62[2] AS 2017#65L]
01 +- Aggregate [city#8], [city#8, pivotfirst(year#9, count(1) AS `count`#54L, 2015, 2
016, 2017, 0, 0) AS __pivot_count(1) AS `count` AS `count(1) AS ``count``#62]
02     +- Aggregate [city#8, year#9], [city#8, year#9, count(1) AS count#24L AS count(1
) AS `count`#54L]
03         +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
04             +- LocalRelation [_1#3, _2#4, _3#5]
```

Creating Pivot Instance

Pivot takes the following when created:

- Grouping [named expressions](#)
- Pivot column [expression](#)
- Pivot values [literals](#)
- Aggregation [expressions](#)
- Child [logical plan](#)

Project Unary Logical Operator

`Project` is a [unary logical operator](#) that takes the following when created:

- Project named expressions
- Child [logical operator](#)

`Project` is [created](#) to represent the following:

- Dataset operators, i.e. [joinWith](#), [select](#) (incl. `selectUntyped`), `unionByName`
- `KeyValueGroupedDataset` operators, i.e. `keys`, `mapValues`
- `CreateViewCommand` logical command is [executed](#) (and `aliasPlan`)
- SQL's [SELECT](#) queries with named expressions

`Project` can also appear in a logical plan after [analysis](#) or [optimization](#) phases.

```
// FIXME Add examples for the following operators
// Dataset.unionByName
// KeyValueGroupedDataset.mapValues
// KeyValueGroupedDataset.keys
// CreateViewCommand.aliasPlan

// joinWith operator
case class Person(id: Long, name: String, cityId: Long)
case class City(id: Long, name: String)
val family = Seq(
  Person(0, "Agata", 0),
  Person(1, "Iweta", 0),
  Person(2, "Patryk", 2),
  Person(3, "Maksym", 0)).toDS
val cities = Seq(
  City(0, "Warsaw"),
  City(1, "Washington"),
  City(2, "Sopot")).toDS
val q = family.joinWith(cities, family("cityId") === cities("id"), "inner")
scala> println(q.queryExecution.logical.numberedTreeString)
00 Join Inner, (_1#41.cityId = _2#42.id)
01 :- Project [named_struct(id, id#32L, name, name#33, cityId, cityId#34L) AS _1#41]
02 : +- LocalRelation [id#32L, name#33, cityId#34L]
03 +- Project [named_struct(id, id#38L, name, name#39) AS _2#42]
04   +- LocalRelation [id#38L, name#39]

// select operator
val qs = spark.range(10).select($"id")
scala> println(qs.queryExecution.logical.numberedTreeString)
```

```

00 'Project [unresolvedalias('id, None)]
01 +- Range (0, 10, step=1, splits=Some(8))

// select[U1](c1: TypedColumn[T, U1])
scala> :type q
org.apache.spark.sql.Dataset[(Person, City)]

val left = "$_1".as[Person]
val ql = q.select(left)
scala> println(ql.queryExecution.logical.numberedTreeString)
00 'SerializeFromObject [assertnonnull(assertnonnull(input[0, $line14.$read$$iw$$iw$Person, true])).id AS id#87L, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnonnull(assertnonnull(input[0, $line14.$read$$iw$$iw$Person, true])).name, true, false) AS name#88, assertnonnull(assertnonnull(input[0, $line14.$read$$iw$$iw$Person, true])).cityId AS cityId#89L]
01 +- 'MapElements <function1>, class scala.Tuple1, [StructField(_1, StructType(StructField(id, LongType, false), StructField(name, StringType, true), StructField(cityId, LongType, false)), true)], obj#86: $line14.$read$$iw$$iw$Person
02     +- 'DeserializeToObject unresolveddeserializer(newInstance(class scala.Tuple1)),
obj#85: scala.Tuple1
03         +- Project [_1#44]
04             +- Join Inner, (_1#44.cityId = _2#45.id)
05                 :- Project [named_struct(id, id#32L, name, name#33, cityId, cityId#34L)
AS _1#44]
06                     : +- LocalRelation [id#32L, name#33, cityId#34L]
07                     +- Project [named_struct(id, id#38L, name, name#39) AS _2#45]
08                         +- LocalRelation [id#38L, name#39]

// SQL
spark.range(10).createOrReplaceTempView("nums")
val qn = spark.sql("select * from nums")
scala> println(qn.queryExecution.logical.numberedTreeString)
00 'Project [*]
01 +- 'UnresolvedRelation `nums`

// Examples with Project that was added during analysis
// Examples with Project that was added during optimization

```

Note

Nondeterministic expressions are allowed in `Project` logical operator and enforced by `CheckAnalysis`.

The output schema of a `Project` is...FIXME

`maxRows` ...FIXME

`resolved` ...FIXME

`validConstraints` ...FIXME

Use `select` operator from [Catalyst DSL](#) to create a `Project` logical operator, e.g. for testing or Spark SQL internals exploration.

Tip

```
import org.apache.spark.sql.catalyst.dsl.plans._ // <-- gives table and select
import org.apache.spark.sql.catalyst.dsl.expressions.star
val plan = table("a").select(star())
scala> println(plan.numberedTreeString)
00 'Project [*]
01 +- 'UnresolvedRelation `a`
```

Range Leaf Logical Operator

Range is a [leaf logical operator](#) that...FIXME

Repartition Logical Operators — Repartition and RepartitionByExpression

[Repartition](#) and [RepartitionByExpression](#) (**repartition operations** in short) are unary logical operators that create a new `RDD` that has exactly `numPartitions` partitions.

Note

`RepartitionByExpression` is also called **distribute** operator.

[Repartition](#) is the result of [coalesce](#) or [repartition](#) (with no partition expressions defined) operators.

```
val rangeAlone = spark.range(5)

scala> rangeAlone.rdd.getNumPartitions
res0: Int = 8

// Repartition the records

val withRepartition = rangeAlone.repartition(numPartitions = 5)

scala> withRepartition.rdd.getNumPartitions
res1: Int = 5

scala> withRepartition.explain(true)
== Parsed Logical Plan ==
Repartition 5, true
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Exchange RoundRobinPartitioning(5)
+- *Range (0, 5, step=1, splits=Some(8))

// Coalesce the records

val withCoalesce = rangeAlone.coalesce(numPartitions = 5)
scala> withCoalesce.explain(true)
== Parsed Logical Plan ==
Repartition 5, false
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Coalesce 5
+- *Range (0, 5, step=1, splits=Some(8))
```

[RepartitionByExpression](#) is the result of the following operators:

- [Dataset.repartition](#) operator (with explicit partition expressions defined)
- [Dataset.repartitionByRange](#)
- [DISTRIBUTE BY](#) SQL clause.

```
// RepartitionByExpression
// 1) Column-based partition expression only
scala> rangeAlone.repartition(partitionExprs = 'id % 2).explain(true)
== Parsed Logical Plan ==
'RepartitionByExpression [('id % 2)], 200
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Exchange hashpartitioning((id#10L % 2), 200)
+- *Range (0, 5, step=1, splits=Some(8))

// 2) Explicit number of partitions and partition expression
scala> rangeAlone.repartition(numPartitions = 2, partitionExprs = 'id % 2).explain(true)
)
== Parsed Logical Plan ==
'RepartitionByExpression [('id % 2)], 2
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Exchange hashpartitioning((id#10L % 2), 2)
+- *Range (0, 5, step=1, splits=Some(8))
```

`Repartition` and `RepartitionByExpression` logical operators are described by:

- `shuffle` flag
- target number of partitions

Note	BasicOperators strategy resolves <code>Repartition</code> to ShuffleExchangeExec (with RoundRobinPartitioning partitioning scheme) or CoalesceExec physical operators per shuffle — enabled or not, respectively.
------	---

Note	BasicOperators strategy resolves <code>RepartitionByExpression</code> to ShuffleExchangeExec physical operator with HashPartitioning partitioning scheme.
------	---

Repartition Operation Optimizations

- [CollapseRepartition](#) logical optimization collapses adjacent repartition operations.
- Repartition operations allow [FoldablePropagation](#) and [PushDownPredicate](#) logical optimizations to "push through".
- [PropagateEmptyRelation](#) logical optimization may result in an empty [LocalRelation](#) for repartition operations.

ResolvedHint Unary Logical Operator

ResolvedHint is a [unary logical operator](#) that...FIXME

ResolvedHint is [created](#) when...FIXME

When requested for [output schema](#), ResolvedHint uses the output of the child logical operator.

ResolvedHint simply requests the [child logical operator](#) for the [canonicalized version](#).

Creating ResolvedHint Instance

ResolvedHint takes the following when created:

- Child [logical operator](#)
- [Query hints](#)

SaveIntoDataSourceCommand Logical Command

`SaveIntoDataSourceCommand` is a [logical command](#) that, when [executed](#), [FIXME](#).

`SaveIntoDataSourceCommand` is [created](#) exclusively when `DataSource` is requested to [create a logical command for writing](#) (to a [CreatableRelationProvider](#) data source).

`SaveIntoDataSourceCommand` returns the [logical query plan](#) when requested for the [inner nodes](#) (that should be shown as an inner nested tree of this node).

```
// DEMO Example with inner nodes that should be shown as an inner nested tree of this node

val lines = Seq("SaveIntoDataSourceCommand").toDF("line")

// NOTE: There are two CreatableRelationProviders: jdbc and kafka
// jdbc is simpler to use in spark-shell as it does not need --packages
val url = "jdbc:derby:memory:;databaseName=/tmp/test;create=true"
val requiredOpts = Map("url" -> url, "dbtable" -> "lines")
// Use overwrite SaveMode to make the demo reproducible
import org.apache.spark.sql.SaveMode.Overwrite
lines.write.options(requiredOpts).format("jdbc").mode(Overwrite).save

// Go to web UI's SQL tab and see the last executed query
```

`SaveIntoDataSourceCommand` [redacts](#) the [options](#) for the [simple description](#) with state prefix.

```
SaveIntoDataSourceCommand [dataSource], [redacted], [mode]
```

Executing Logical Command — `run` Method

```
run(
  sparkSession: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (`run`) a logical command.

`run` simply requests the [CreatableRelationProvider](#) data source to [save](#) the rows of a structured query (a [DataFrame](#)).

In the end, `run` returns an empty `seq[Row]` (just to follow the signature and please the Scala compiler).

Creating SaveIntoDataSourceCommand Instance

`SaveIntoDataSourceCommand` takes the following when created:

- `Logical query plan`
- `CreatableRelationProvider` data source
- Options (as `Map[String, String]`)
- `SaveMode`

ShowCreateTableCommand Logical Command

`ShowCreateTableCommand` is a logical command that executes a `SHOW CREATE TABLE` SQL statement (with a data source / non-Hive or a Hive table).

`ShowCreateTableCommand` is created when `SparkSqlAstBuilder` is requested to parse `SHOW CREATE TABLE` SQL statement.

`ShowCreateTableCommand` uses a single `createtab_stmt` column (of type `StringType`) for the output schema.

```
import org.apache.spark.sql.SaveMode
spark.range(10e4.toLong)
  .write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode(SaveMode.Overwrite)
  .saveAsTable("bucketed_4_10e4")
scala> sql("SHOW CREATE TABLE bucketed_4_10e4").show(truncate = false)
+-----+
|createtab_stmt
|-----+
|-----+
|-----+
|CREATE TABLE `bucketed_4_10e4`(`id` BIGINT)
USING parquet
OPTIONS (
  `serialization.format` '1'
)
CLUSTERED BY (id)
SORTED BY (id)
INTO 4 BUCKETS
|
+-----+
|-----+
|-----+
|-----+
scala> sql("SHOW CREATE TABLE bucketed_4_10e4").as[String].collect.foreach(println)
CREATE TABLE `bucketed_4_10e4`(`id` BIGINT)
USING parquet
OPTIONS (
  `serialization.format` '1'
)
CLUSTERED BY (id)
SORTED BY (id)
INTO 4 BUCKETS
```

`ShowCreateTableCommand` takes a single `TableIdentifier` when created.

Executing Logical Command — `run` Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note

`run` is part of [RunnableCommand Contract](#) to execute (run) a logical command.

`run` requests the `sparkSession` for the [SessionState](#) that is used to access the [SessionCatalog](#).

`run` then requests the `SessionCatalog` to retrieve the table metadata from the external catalog (metastore).

`run` then [showCreateDataSourceTable](#) for a data source / non-Hive table or [showCreateHiveTable](#) for a Hive table (per the `table metadata`).

In the end, `run` returns the `CREATE TABLE` statement in a single `Row`.

showHiveTableNonDataColumns Internal Method

```
showHiveTableNonDataColumns(metadata: CatalogTable, builder: StringBuilder): Unit
```

`showHiveTableNonDataColumns` ...FIXME

Note

`showHiveTableNonDataColumns` is used exclusively when `ShowCreateTableCommand` logical command is requested to [showCreateHiveTable](#).

showCreateHiveTable Internal Method

```
showCreateHiveTable(metadata: CatalogTable): String
```

`showCreateHiveTable` ...FIXME

Note

`showCreateHiveTable` is used exclusively when `ShowCreateTableCommand` logical command is executed (with a Hive `table`).

showHiveTableHeader Internal Method

```
showHiveTableHeader(metadata: CatalogTable, builder: StringBuilder): Unit
```

showHiveTableHeader ...FIXME

Note

showHiveTableHeader is used exclusively when ShowCreateTableCommand logical command is requested to [showCreateHiveTable](#).

ShowTablesCommand Logical Command

ShowTablesCommand is a [logical command](#) for...FIXME

Executing Logical Command — run Method

```
run(sparkSession: SparkSession): Seq[Row]
```

Note

run is part of [RunnableCommand Contract](#) to execute (run) a logical command.

run ...FIXME

Sort Unary Logical Operator

`Sort` is a [unary logical operator](#) that represents the following in a logical plan:

- `ORDER BY`, `SORT BY`, `SORT BY ... DISTRIBUTE BY` and `CLUSTER BY` clauses (when `AstBuilder` is requested to [parse a query](#))
- `Dataset.sortWithinPartitions`, `Dataset.sort` and `Dataset.randomSplit` operators

```
// Using the feature of ordinal literal
val ids = Seq(1,3,2).toDF("id").sort(lit(1))
val logicalPlan = ids.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 Sort [1 ASC NULLS FIRST], true
01 +- AnalysisBarrier
02     +- Project [value#22 AS id#24]
03         +- LocalRelation [value#22]

import org.apache.spark.sql.catalyst.plans.logical.Sort
val sortOp = logicalPlan.collect { case s: Sort => s }.head
scala> println(sortOp.numberedTreeString)
00 Sort [1 ASC NULLS FIRST], true
01 +- AnalysisBarrier
02     +- Project [value#22 AS id#24]
03         +- LocalRelation [value#22]
```

```
val nums = Seq((0, "zero"), (1, "one")).toDF("id", "name")
// Creates a Sort logical operator:
// - descending sort direction for id column (specified explicitly)
// - name column is wrapped with ascending sort direction
val numsOrdered = nums.sort('id.desc, 'name)
val logicalPlan = numsOrdered.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Sort ['id DESC NULLS LAST, 'name ASC NULLS FIRST], true
01 +- Project [_1#11 AS id#14, _2#12 AS name#15]
02     +- LocalRelation [_1#11, _2#12]
```

`Sort` takes the following when created:

- `SortOrder` ordering expressions
- `global` flag for global (`true`) or partition-only (`false`) sorting
- Child [logical plan](#)

The [output schema](#) of a `Sort` operator is the output of the [child](#) logical operator.

The `maxRows` of a `sort` operator is the `maxRows` of the `child` logical operator.

Tip	Use <code>orderBy</code> or <code>sortBy</code> operators from the Catalyst DSL to create a <code>sort</code> logical operator, e.g. for testing or Spark SQL internals exploration.
-----	--

Note	Sorting is supported for columns of orderable type only (which is enforced at analysis when <code>CheckAnalysis</code> is requested to <code>checkAnalysis</code>).
------	--

Note	<code>sort</code> logical operator is resolved to <code>SortExec</code> unary physical operator when BasicOperators execution planning strategy is executed.
------	--

Catalyst DSL — `orderBy` and `sortBy` Operators

```
orderBy(sortExprs: SortOrder*): LogicalPlan
sortBy(sortExprs: SortOrder*): LogicalPlan
```

`orderBy` and `sortBy` [create a `Sort` logical operator with the `global` flag on and off](#), respectively.

```
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")

import org.apache.spark.sql.catalyst.dsl.expressions._
val globalSortById = t1.orderBy('id.asc_nullsLast)

// Note true for the global flag
scala> println(globalSortById.numberedTreeString)
00 'Sort ['id ASC NULLS LAST], true
01 +- 'UnresolvedRelation `t1`

// Note false for the global flag
val partitionOnlySortById = t1.sortBy('id.asc_nullsLast)
scala> println(partitionOnlySortById.numberedTreeString)
00 'Sort ['id ASC NULLS LAST], false
01 +- 'UnresolvedRelation `t1`
```

SubqueryAlias Unary Logical Operator

`SubqueryAlias` is a [unary logical operator](#) that represents an [aliased subquery](#) (i.e. the `child` logical query plan with the `alias` in the [output schema](#)).

`SubqueryAlias` is [created](#) when:

- `AstBuilder` is requested to parse a [named](#) or [aliased](#) query, [aliased query plan](#) and `mayApplyAliasPlan` in a SQL statement
- `Dataset.as` operator is used
- `SessionCatalog` is requested to [find a table or view in catalogs](#)
- `RewriteCorrelatedScalarSubquery` logical optimization is requested to `constructLeftJoins` (when [applied](#) to [Aggregate](#), [Project](#) or [Filter](#) logical operators with correlated scalar subqueries)

`SubqueryAlias` simply requests the `child` logical operator for the [canonicalized version](#).

When requested for [output schema attributes](#), `SubqueryAlias` requests the `child` logical operator for them and adds the `alias` as a [qualifier](#).

Note

[EliminateSubqueryAliases](#) logical optimization eliminates (removes) `SubqueryAlias` operators from a logical query plan.

Note

[RewriteCorrelatedScalarSubquery](#) logical optimization rewrites correlated scalar subqueries with `SubqueryAlias` operators.

Catalyst DSL — `subquery` And `as` Operators

```
as(alias: String): LogicalPlan
subquery(alias: Symbol): LogicalPlan
```

`subquery` and `as` operators in Catalyst DSL create a `SubqueryAlias` logical operator, e.g. for testing or Spark SQL internals exploration.

```
import org.apache.spark.sql.catalyst.dsl.plans._  
val t1 = table("t1")  
  
val plan = t1.subquery('a)  
scala> println(plan.numberedTreeString)  
00 'SubqueryAlias a  
01 +- 'UnresolvedRelation `t1`  
  
val plan = t1.as("a")  
scala> println(plan.numberedTreeString)  
00 'SubqueryAlias a  
01 +- 'UnresolvedRelation `t1`
```

Creating SubqueryAlias Instance

`SubqueryAlias` takes the following when created:

- Alias
- Child [logical plan](#)

TypedFilter Logical Operator

TypedFilter is...FIXME

Union Logical Operator

Union is...FIXME

UnresolvedCatalogRelation Leaf Logical Operator — Placeholder of Catalog Tables

`UnresolvedCatalogRelation` is a [leaf logical operator](#) that acts as a placeholder in a logical query plan until [FindDataSourceTable](#) logical evaluation rule resolves it to a concrete relation logical plan (i.e. a [LogicalRelation](#) for a data source table or a [HiveTableRelation](#) for hive table).

`UnresolvedCatalogRelation` is [created](#) when `SessionCatalog` is requested to [find a relation](#) (for [DescribeTableCommand](#) logical command or [ResolveRelations](#) logical evaluation rule).

```
// non-temporary global or local view
// database defined
// externalCatalog.getTable returns non-VIEW table
// Make the example reproducible
val tableName = "t1"
spark.sharedState.externalCatalog.dropTable(
  db = "default",
  table = tableName,
  ignoreIfNotExists = true,
  purge = true)
spark.range(10).write.saveAsTable(tableName)

scala> :type spark.sessionState.catalog
org.apache.spark.sql.catalyst.catalog.SessionCatalog

import org.apache.spark.sql.catalyst.TableIdentifier
val plan = spark.sessionState.catalog.lookupRelation(TableIdentifier(tableName))
scala> println(plan.numberedTreeString)
00 'SubqueryAlias t1
01 +- 'UnresolvedCatalogRelation `default`.`t1`, org.apache.hadoop.hive.ql.io.parquet.
serde.ParquetHiveSerDe
```

When [created](#), `UnresolvedCatalogRelation` asserts that the database is specified.

`UnresolvedCatalogRelation` can never be [resolved](#) and is converted to a [LogicalRelation](#) for a data source table or a [HiveTableRelation](#) for hive table at [analysis phase](#).

`UnresolvedCatalogRelation` uses an empty [output schema](#).

`UnresolvedCatalogRelation` takes a single [CatalogTable](#) when created.

UnresolvedHint Unary Logical Operator — Attaching Hint to Logical Plan

`UnresolvedHint` is a unary logical operator that represents a hint (by [name](#) and [parameters](#)) for the [child](#) logical plan.

`UnresolvedHint` is [created](#) and added to a [logical plan](#) when:

- `Dataset_hint` operator is used
- `AstBuilder` [converts](#) `/*+ hint */` in `SELECT` SQL queries

```
// Dataset API
val q = spark.range(1).hint("myHint", 100, true)
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- Range (0, 1, step=1, splits=Some(8))

// SQL
val q = sql("SELECT /*+ myHint (100, true) */ 1")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- 'Project [unresolvedalias(1, None)]
02     +- OneRowRelation
```

When created `UnresolvedHint` takes:

- Name of a hint
- Parameters of a hint
- Child [logical plan](#)

`UnresolvedHint` can never be [resolved](#) and is supposed to be converted to a [ResolvedHint](#) unary logical operator during [query analysis](#) (or simply removed from a logical plan).

There are the following logical rules that [Spark Analyzer](#) uses to analyze logical plans with the [UnresolvedHint](#) logical operator:

- Note
1. [ResolveBroadcastHints](#) resolves `UnresolvedHint` operators with `BROADCAST`, `BROADCASTJOIN`, `MAPJOIN` hints to a [ResolvedHint](#)
 2. [ResolveCoalesceHints](#) resolves `UnresolvedHint` logical operators with `COALESCE` or `REPARTITION` hints
 3. `RemoveAllHints` simply removes all `UnresolvedHint` operators

The order of executing the above rules matters.

```
// Let's hint the query twice
// The order of hints matters as every hint operator executes Spark analyzer
// That will resolve all but the last hint
val q = spark.range(100).
  hint("broadcast").
  hint("myHint", 100, true)
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- ResolvedHint (broadcast)
02   +- Range (0, 100, step=1, splits=Some(8))

// Let's resolve unresolved hints
import org.apache.spark.sql.catalyst.rules.RuleExecutor
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.analysis.ResolveHints
import org.apache.spark.sql.internal.SQLConf
object HintResolver extends RuleExecutor[LogicalPlan] {
  lazy val batches =
    Batch("Hints", FixedPoint(maxIterations = 100),
      new ResolveHints.ResolveBroadcastHints(SQLConf.get),
      ResolveHints.RemoveAllHints) :: Nil
}
val resolvedPlan = HintResolver.execute(plan)
scala> println(resolvedPlan.numberedTreeString)
00 ResolvedHint (broadcast)
01 +- Range (0, 100, step=1, splits=Some(8))
```

`UnresolvedHint` uses the [child](#) operator's output schema for yours.

Use `hint` operator from [Catalyst DSL](#) to create a `UnresolvedHint` logical operator, e.g. for testing or Spark SQL internals exploration.

Tip

```
// Create a logical plan to add hint to
import org.apache.spark.sql.catalyst.plans.logical.LocalRelation
val r1 = LocalRelation('a.int, 'b.timestamp, 'c.boolean)
scala> println(r1.numberedTreeString)
00 LocalRelation <empty>, [a#0, b#1, c#2]

// Attach hint to the plan
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = r1.hint(name = "myHint", 100, true)
scala> println(plan.numberedTreeString)
00 'UnresolvedHint myHint, [100, true]
01 +- LocalRelation <empty>, [a#0, b#1, c#2]
```

UnresolvedInlineTable Logical Operator

`UnresolvedInlineTable` is a [unary logical operator](#) that represents an inline table (aka *virtual table* in Apache Hive).

`UnresolvedInlineTable` is [created](#) when `AstBuilder` is requested to [parse an inline table](#) in a SQL statement.

```
// `tableAlias` undefined so columns default to `colN`
val q = sql("VALUES 1, 2, 3")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'UnresolvedInlineTable [col1], [List(1), List(2), List(3)]

// CreateNamedStruct with `tableAlias`
val q = sql("VALUES (1, 'a'), (2, 'b') AS t1(a, b)")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'SubqueryAlias t1
01 +- 'UnresolvedInlineTable [a, b], [List(1, a), List(2, b)]
```

`UnresolvedInlineTable` is never [resolved](#) (and is converted to a [LocalRelation](#) in [ResolveInlineTables](#) logical resolution rule).

`UnresolvedInlineTable` uses no [output schema attributes](#).

`UnresolvedInlineTable` uses `expressionsResolved` flag that is on (`true`) only when all the Catalyst expressions in the `rows` are resolved.

Creating UnresolvedInlineTable Instance

`UnresolvedInlineTable` takes the following when created:

- Column names
- Rows (as [Catalyst expressions](#) for every row, i.e. `Seq[Seq[Expression]]`)

UnresolvedRelation Leaf Logical Operator for Table Reference

`UnresolvedRelation` is a [leaf logical operator](#) to represent a **table reference** in a logical query plan that has yet to be resolved (i.e. looked up in a catalog).

Note

If after [Analyzer](#) has finished analyzing a logical query plan the plan has still a `UnresolvedRelation` it [fails the analyze phase](#) with the following `AnalysisException`:

```
Table or view not found: [tableIdentifier]
```

`UnresolvedRelation` is [created](#) when:

- `SparkSession` is requested to [create a DataFrame from a table](#)
- `DataFrameWriter` is requested to [insert a DataFrame into a table](#)
- `INSERT INTO (TABLE)` or `INSERT OVERWRITE TABLE` SQL commands are [executed](#)
- `CreateHiveTableAsSelectCommand` command is executed

Tip

Use `table` operator from [Catalyst DSL](#) to create a `UnresolvedRelation` logical operator, e.g. for testing or Spark SQL internals exploration.

```
import org.apache.spark.sql.catalyst.dsl.plans._  
val plan = table(db = "myDB", ref = "t1")  
scala> println(plan.numberedTreeString)  
00 'UnresolvedRelation `myDB`.`t1`'
```

Note

`UnresolvedRelation` is resolved to...FIXME

UnresolvedTableValuedFunction

UnresolvedTableValuedFunction is...FIXME

Window Unary Logical Operator

`Window` is a [unary logical operator](#) that...FIXME

`Window` is [created](#) when:

- `ExtractWindowExpressions` logical resolution rule is [executed](#)
- `CleanupAliases` logical analysis rule is [executed](#)

When requested for [output schema attributes](#), `Window` requests the [child](#) logical operator for them and adds the [attributes](#) of the `window named expressions`.

Note	<code>Window</code> logical operator is a subject of pruning unnecessary window expressions in ColumnPruning logical optimization and collapsing window operators in CollapseWindow logical optimization.
------	---

Note	<code>Window</code> logical operator is resolved to a WindowExec in BasicOperators execution planning strategy.
------	---

Catalyst DSL — `window` Operator

```
window(
    windowExpressions: Seq[NamedExpression],
    partitionSpec: Seq[Expression],
    orderSpec: Seq[SortOrder]): LogicalPlan
```

`window` operator in Catalyst DSL creates a [Window](#) logical operator, e.g. for testing or Spark SQL internals exploration.

```
// FIXME: DEMO
```

Creating Window Instance

`Window` takes the following when created:

- Window [named expressions](#)
- Window partition specification [expressions](#)
- Window order specification (as a collection of `SortOrder` [expressions](#))
- Child [logical operator](#)

Creating AttributeSet with Window Expression Attributes

— `windowOutputSet` Method

```
windowOutputSet: AttributeSet
```

`windowOutputSet` simply creates a `AttributeSet` with the [attributes](#) of the `window` named [expressions](#).

Note

`windowOutputSet` is used when:

- `ColumnPruning` logical optimization is [executed](#) (on a `Project` operator with a `Window` as the [child operator](#))
- `CollapseWindow` logical optimization is [executed](#) (on a `window` operator with another `Window` operator as the [child](#))

WithWindowDefinition Unary Logical Operator

`WithWindowDefinition` is a [unary logical plan](#) with a single `child` logical plan and a `windowDefinitions` lookup table of [WindowSpecDefinition](#) per name.

`WithWindowDefinition` is created exclusively when `AstBuilder` [parses window definitions](#).

The [output schema](#) of `WithWindowDefinition` is exactly the output attributes of the `child` logical operator.

```
// Example with window specification alias and definition
val sqlText = """
    SELECT count(*) OVER anotherWindowSpec
    FROM range(5)
    WINDOW
        anotherWindowSpec AS myWindowSpec,
        myWindowSpec AS (
            PARTITION BY id
            RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
        )
"""
import spark.sessionState.{analyzer, sqlParser}
val parsedPlan = sqlParser.parsePlan(sqlText)

scala> println(parsedPlan.numberedTreeString)
00 'WithWindowDefinition Map(anotherWindowSpec -> windowspecdefinition('id, RANGE BETW
EEN UNBOUNDED PRECEDING AND CURRENT ROW), myWindowSpec -> windowspecdefinition('id, RA
NGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW))
01 +- 'Project [unresolvedalias(unresolvedwindowexpression('count(1), WindowSpecRefere
nce(anotherWindowSpec)), None)]
02   +- 'UnresolvedTableValuedFunction range, [5]

val plan = analyzer.execute(parsedPlan)
scala> println(plan.numberedTreeString)
00 Project [count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDING AND CURRE
NT ROW)#75L]
01 +- Project [id#73L, count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDIN
G AND CURRENT ROW)#75L, count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDI
NG AND CURRENT ROW)#75L]
02   +- Window [count(1) windowspecdefinition(id#73L, RANGE BETWEEN UNBOUNDED PRECEDI
NG AND CURRENT ROW) AS count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDI
NG AND CURRENT ROW)#75L], [id#73L]
03     +- Project [id#73L]
04       +- Range (0, 5, step=1, splits=None)
```


WriteToDataSourceV2 Logical Operator — Writing Data to DataSourceV2

`WriteToDataSourceV2` is a [logical operator](#) that represents writing data to a [DataSourceV2](#) data source in the [Data Source API V2](#).

Note

`WriteToDataSourceV2` is deprecated for [AppendData](#) logical operator since Spark SQL 2.4.0.

`WriteToDataSourceV2` is [created](#) when:

- `DataFrameWriter` is requested to save a `DataFrame` to a [data source](#) (that is a [DataSourceV2](#) data source with [WriteSupport](#))
- Spark Structured Streaming's `MicroBatchExecution` is requested to run a streaming batch (with a streaming sink with `StreamWriterSupport`)

`WriteToDataSourceV2` takes the following to be created:

- [DataSourceWriter](#)
- Child [logical plan](#)

When requested for the [child operators](#), `WriteToDataSourceV2` gives the one [child logical plan](#).

When requested for the [output attributes](#), `WriteToDataSourceV2` gives no attributes (an empty collection).

`WriteToDataSourceV2` is planned (*translated*) to a [WriteToDataSourceV2Exec](#) physical operator (when [DataSourceV2Strategy](#) execution planning strategy is requested to [plan a logical query](#)).

View Unary Logical Operator

`View` is a [logical operator](#) with a single `child` logical operator.

`View` is [created](#) exclusively when `SessionCatalog` is requested to [find a relation in the catalogs](#) (e.g. when `DescribeTableCommand` logical command is [executed](#) and the table type is `VIEW`).

```
// Let's create a view first
// Using SQL directly to manage views is so much nicer
val name = "demo_view"
sql(s"CREATE OR REPLACE VIEW $name COMMENT 'demo view' AS VALUES 1,2")
assert(spark.catalog.tableExists(name))

val q = sql(s"DESC EXTENDED $name")

val allRowsIncluded = 100
scala> q.show(numRows = allRowsIncluded)
+-----+-----+-----+
|       col_name|      data_type|comment|
+-----+-----+-----+
|          col1|          int|    null|
|           |           |   |
| # Detailed Table ...|           |   |
|          Database|      default|   |
|          Table|      demo_view|   |
|          Owner|        jacek|   |
|     Created Time|Thu Aug 30 08:55:...|   |
|     Last Access|Thu Jan 01 01:00:...|   |
|     Created By|      Spark 2.3.1|   |
|          Type|        VIEW|   |
|     Comment|      demo view|   |
|     View Text|      VALUES 1,2|   |
|View Default Data...|      default|   |
|View Query Output...|[col1]|   |
|  Table Properties|[transient_lastDd...|   |
|     Serde Library|org.apache.hadoop...|   |
|     InputFormat|org.apache.hadoop...|   |
|     OutputFormat|org.apache.hadoop...|   |
| Storage Properties|[serialization.fo...|   |
+-----+-----+-----+
```

`View` is a [MultiInstanceRelation](#) so a new instance will be created to appear multiple times in a physical query plan. When requested for a new instance, `View` creates new instances of the [output attributes](#).

`View` is considered resolved only when the `child` is.

`View` has the following simple description (with state prefix):

```
View ([identifier], [output])
```

```
val name = "demo_view"
sql(s"CREATE OR REPLACE VIEW $name COMMENT 'demo view' AS VALUES 1,2")
assert(spark.catalog.tableExists(name))

val q = spark.table(name)
val qe = q.queryExecution

val logicalPlan = qe.logical
scala> println(logicalPlan.simpleString)
'UnresolvedRelation `demo_view`

val analyzedPlan = qe.analyzed
scala> println(analyzedPlan.numberedTreeString)
00 SubqueryAlias demo_view
01 +- View (`default`.`demo_view`, [col1#33])
02   +- Project [cast(col1#34 as int) AS col1#33]
03     +- LocalRelation [col1#34]

// Skip SubqueryAlias
scala> println(analyzedPlan.children.head.simpleString)
View (`default`.`demo_view`, [col1#33])
```

Note

`View` is resolved by [ResolveRelations](#) logical resolution.

Note

[AliasViewChild](#) logical analysis rule makes sure that the `output` of a `View` matches the output of the `child` logical operator.

Note

[EliminateView](#) logical optimization removes (eliminates) `View` operators from a logical query plan.

Note

[Inserting into a view is not allowed.](#)

Creating View Instance

`View` takes the following when created:

- [CatalogTable](#)
- [Output schema attributes](#) (as `Seq[Attribute]`)
- Child [logical plan](#)

SparkPlan Contract — Physical Operators in Physical Query Plan of Structured Query

`SparkPlan` is the [contract](#) of **physical operators** to build a **physical query plan** (aka *query execution plan*).

`SparkPlan` contract requires that a concrete physical operator implements [doExecute](#) method.

```
doExecute(): RDD[InternalRow]
```

`doExecute` allows a physical operator to describe a distributed computation (that is a runtime representation of the operator in particular and a structured query in general) as an RDD of [internal binary rows](#), i.e. `RDD[InternalRow]` , and thus [execute](#).

Table 1. SparkPlan's Extension Hooks

Name	Description
doExecuteBroadcast	<p>By default reports a <code>UnsupportedOperationException</code>.</p> <p><code>[nodeName]</code> does not implement <code>doExecuteBroadcast</code></p> <p>Executed exclusively as part of <code>executeBroadcast</code> to return the result of a structured query as a broadcast variable.</p>
doPrepare	<p>Prepares a physical operator for execution</p> <p>Executed exclusively as part of <code>prepare</code> and is supposed to set some state up before executing a query (e.g. <code>BroadcastExchangeExec</code> to broadcast a relation asynchronously or <code>SubqueryExec</code> to execute a child operator)</p>
requiredChildDistribution	<p><code>requiredChildDistribution: Seq[Distribution]</code></p> <p>The required partition requirements (aka child output distributions) of the input data, i.e. how <code>children</code> physical operators' output is split across partitions.</p> <p>Defaults to a <code>UnspecifiedDistribution</code> for all of the <code>child</code> operators.</p> <p>Used exclusively when <code>EnsureRequirements</code> physical query plan optimization is <code>executed</code> (and <code>enforces partition requirements</code>).</p>
requiredChildOrdering	<p><code>requiredChildOrdering: Seq[Seq[SortOrder]]</code></p> <p>Specifies required sort ordering for each partition requirement (from <code>children</code> operators)</p> <p>Defaults to no sort ordering for all of the physical operator's <code>children</code>.</p> <p>Used exclusively when <code>EnsureRequirements</code> physical query plan optimization is <code>executed</code> (and <code>enforces partition requirements</code>).</p>

`SparkPlan` is a recursive data structure in Spark SQL's `Catalyst tree manipulation framework` and as such represents a single **physical operator** in a physical execution query plan as well as a **physical execution query plan** itself (i.e. a tree of physical operators in a query plan of a structured query).

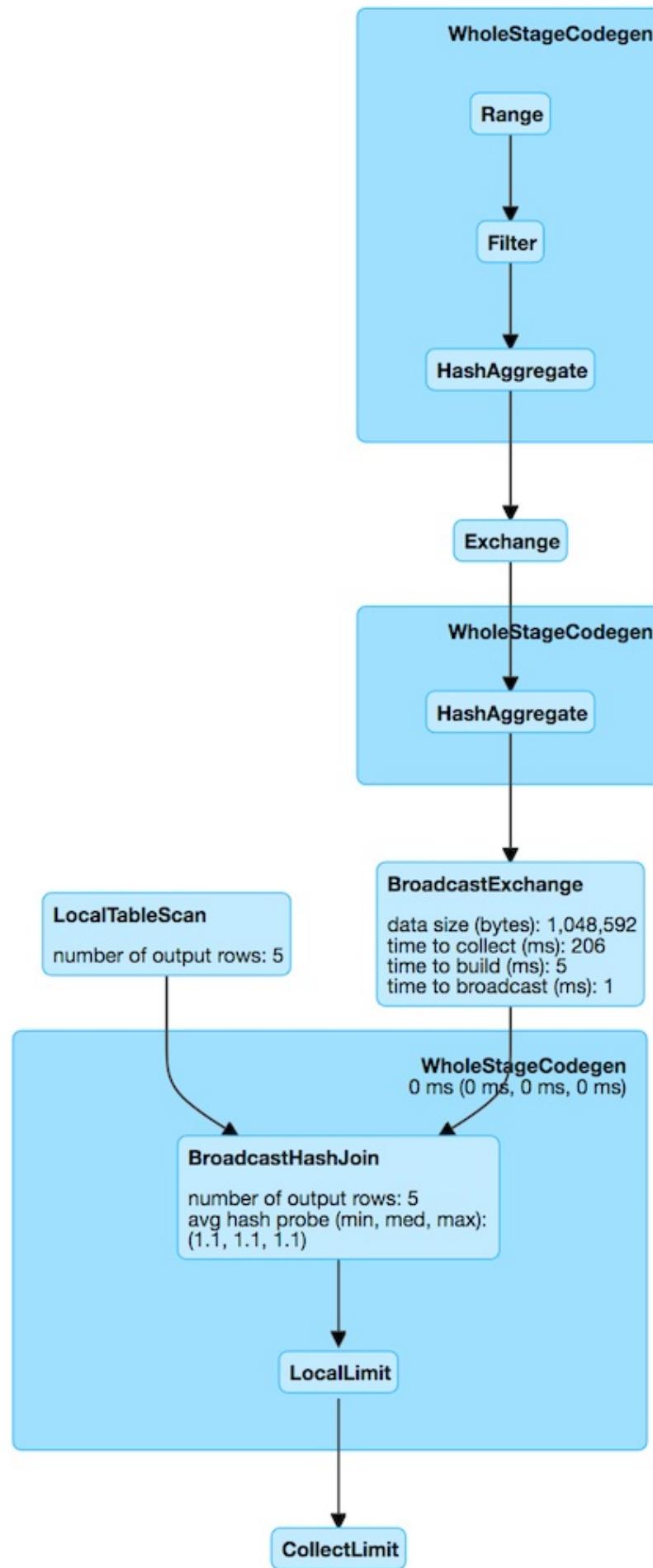


Figure 1. Physical Plan of Structured Query (i.e. Tree of SparkPlans)

Note	A structured query can be expressed using Spark SQL's high-level Dataset API for Scala, Java, Python, R or good ol' SQL.
------	--

A `sparkPlan` physical operator is a [Catalyst tree node](#) that may have zero or more [child physical operators](#).

Note	A structured query is basically a single <code>sparkPlan</code> physical operator with child physical operators .
------	---

Note	Spark SQL uses Catalyst tree manipulation framework to compose nodes to build a tree of (logical or physical) operators that, in this particular case, is composing <code>sparkPlan</code> physical operator nodes to build the physical execution plan tree of a structured query.
------	---

The entry point to **Physical Operator Execution Pipeline** is [execute](#).

When [executed](#), `sparkPlan` [executes the internal query implementation](#) in a named scope (for visualization purposes, e.g. web UI) that triggers [prepare](#) of the children physical operators first followed by [prepareSubqueries](#) and finally [doPrepare](#) methods. After [subqueries have finished](#), [doExecute](#) method is eventually triggered.

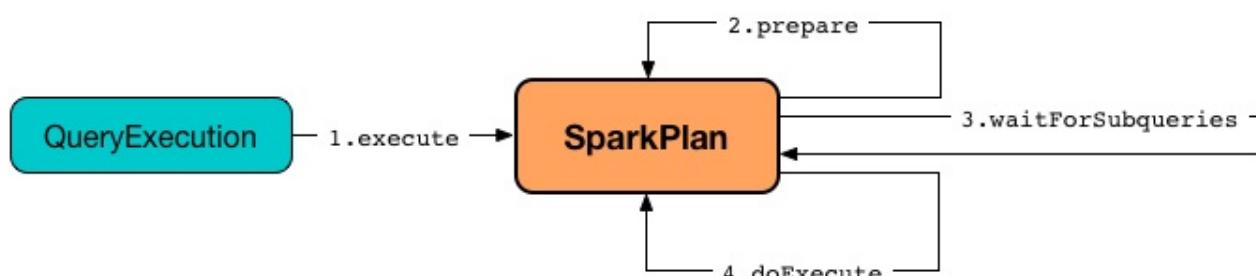


Figure 2. SparkPlan's Execution (execute Method)

The result of [executing](#) a `sparkPlan` is an `RDD` of [internal binary rows](#), i.e.

`RDD[InternalRow]`.

Note	Executing a structured query is simply a translation of the higher-level Dataset-based description to an RDD-based runtime representation that Spark will in the end execute (once an Dataset action is used).
------	--

Caution	FIXME Picture between Spark SQL's Dataset \Rightarrow Spark Core's RDD
---------	--

`SparkPlan` has access to the owning `SparkContext` (from the Spark Core).

	<p><code>execute</code> is called when <code>QueryExecution</code> is requested for the <code>RDD</code> that is Spark Core's physical execution plan (as a <code>RDD</code> lineage) that triggers query execution (i.e. physical planning, but not execution of the plan) and <i>could</i> be considered execution of a structured query.</p>
Note	<p>The <i>could</i> part above refers to the fact that the final execution of a structured query happens only when a <code>RDD</code> action is executed on the <code>RDD</code> of a structured query. And hence the need for Spark SQL's high-level Dataset API in which the Dataset operators simply execute a <code>RDD</code> action on the corresponding <code>RDD</code>. <i>Easy, isn't it?</i></p>

	<p>Use <code>explain</code> operator to see the execution plan of a structured query.</p>
Tip	<pre>val q = // your query here q.explain</pre> <p>You may also access the execution plan of a <code>Dataset</code> using its <code>queryExecution</code> property.</p> <pre>val q = // your query here q.queryExecution.sparkPlan</pre>

The [SparkPlan contract](#) assumes that concrete physical operators define `doExecute` method (with optional hooks like `doPrepare`) which is executed when the physical operator is [executed](#).

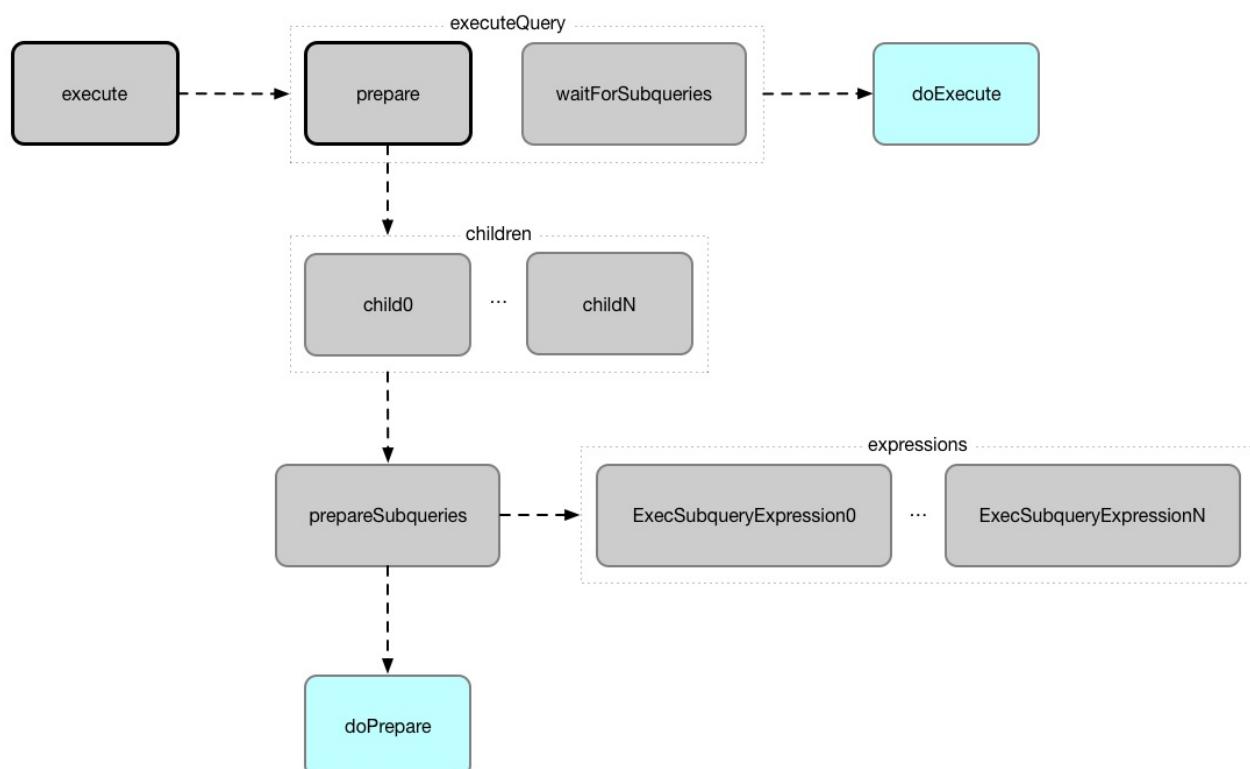


Figure 3. SparkPlan.execute — Physical Operator Execution Pipeline

`SparkPlan` has the following `final` methods that prepare execution environment and pass calls to corresponding methods (that constitute [SparkPlan Contract](#)).

Table 2. SparkPlan's Final Methods

Name	Description		
<code>execute</code>	<pre>execute(): RDD[InternalRow]</pre> <p>"Executes" a physical operator (and its children) that triggers physical query planning and in the end generates an <code>RDD</code> of internal binary rows (i.e. <code>RDD[InternalRow]</code>).</p> <p>Used <i>mostly</i> when <code>QueryExecution</code> is requested for the RDD-based runtime representation of a structured query (that describes a distributed computation using Spark Core's <code>RDD</code>).</p> <p>Internally, <code>execute</code> first prepares the physical operator for execution and eventually requests it to doExecute.</p> <table border="1"> <tr> <td>Note</td><td>Executing <code>doExecute</code> in a named scope happens only after the operator is prepared for execution followed by waiting for any subqueries to finish.</td></tr> </table>	Note	Executing <code>doExecute</code> in a named scope happens only after the operator is prepared for execution followed by waiting for any subqueries to finish .
Note	Executing <code>doExecute</code> in a named scope happens only after the operator is prepared for execution followed by waiting for any subqueries to finish .		
<code>executeQuery</code>	<pre>executeQuery[T](query: => T): T</pre> <p>Executes a physical operator in a single <code>RDD</code> scope, i.e. all <code>RDDs</code> created during execution of the physical operator have the same scope.</p> <p><code>executeQuery</code> executes the input <code>query</code> after the following methods (in order):</p> <ol style="list-style-type: none"> 1. prepare 2. waitForSubqueries <table border="1"> <tr> <td>Note</td><td> <p><code>executeQuery</code> is used when:</p> <ul style="list-style-type: none"> • <code>SparkPlan</code> is executed (in which the input <code>query</code> is just <code>doExecute</code>) • <code>SparkPlan</code> is requested to executeBroadcast (in which the input <code>query</code> is just <code>doExecuteBroadcast</code>) • <code>CodegenSupport</code> is requested for the Java source code of a physical operator (in which the input <code>query</code> is <code>doProduce</code>) </td></tr> </table>	Note	<p><code>executeQuery</code> is used when:</p> <ul style="list-style-type: none"> • <code>SparkPlan</code> is executed (in which the input <code>query</code> is just <code>doExecute</code>) • <code>SparkPlan</code> is requested to executeBroadcast (in which the input <code>query</code> is just <code>doExecuteBroadcast</code>) • <code>CodegenSupport</code> is requested for the Java source code of a physical operator (in which the input <code>query</code> is <code>doProduce</code>)
Note	<p><code>executeQuery</code> is used when:</p> <ul style="list-style-type: none"> • <code>SparkPlan</code> is executed (in which the input <code>query</code> is just <code>doExecute</code>) • <code>SparkPlan</code> is requested to executeBroadcast (in which the input <code>query</code> is just <code>doExecuteBroadcast</code>) • <code>CodegenSupport</code> is requested for the Java source code of a physical operator (in which the input <code>query</code> is <code>doProduce</code>) 		

	<pre>prepare(): Unit</pre> <p>Prepares a physical operator for execution</p> <p><code>prepare</code> is used mainly when a physical operator is requested to execute a structured query</p> <p><code>prepare</code> is also used recursively for every child physical operator (down the physical plan) and when a physical operator is requested to prepare subqueries.</p>		
<code>prepare</code>	<table border="1"> <tr> <td>Note</td><td><code>prepare</code> is idempotent, i.e. can be called multiple times with no change to the final result. It uses <code>prepared</code> internal flag to execute the physical operator once only.</td></tr> </table> <p>Internally, <code>prepare</code> calls doPrepare of its children before prepareSubqueries and doPrepare.</p>	Note	<code>prepare</code> is idempotent, i.e. can be called multiple times with no change to the final result. It uses <code>prepared</code> internal flag to execute the physical operator once only.
Note	<code>prepare</code> is idempotent, i.e. can be called multiple times with no change to the final result. It uses <code>prepared</code> internal flag to execute the physical operator once only.		
<code>executeBroadcast</code>	<pre>executeBroadcast[T](): broadcast.Broadcast[T]</pre> <p>Calls doExecuteBroadcast</p>		

Table 3. Physical Query Operators / Specialized SparkPlans

Name	Description
<code>BinaryExecNode</code>	Binary physical operator with two child <code>left</code> and <code>right</code> physical operators
<code>LeafExecNode</code>	<p>Leaf physical operator with no children</p> <p>By default, the set of all attributes that are produced is exactly the set of attributes that are output.</p>
<code>UnaryExecNode</code>	Unary physical operator with one <code>child</code> physical operator
Note	The naming convention for physical operators in Spark's source code is to have their names end with the <code>Exec</code> prefix, e.g. <code>DebugExec</code> or <code>LocalTableScanExec</code> that is however removed when the operator is displayed, e.g. in web UI .

Table 4. SparkPlan's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
prepared	Flag that controls that prepare is executed only once.
subexpressionEliminationEnabled	Flag that controls whether the subexpression elimination optimization is enabled or not. Used when the following physical operators are requested to execute (i.e. describe a distributed computation as an RDD of internal rows): <ul style="list-style-type: none"> • ProjectExec • HashAggregateExec (and for finishAggregate) • ObjectHashAggregateExec • SortAggregateExec • WindowExec (and creates a lookup table for WindowExpressions and factory functions for WindowFunctionFrame)

Caution

`FIXME` `SparkPlan` is `Serializable`. Why? Is this because `Dataset.cache` persists executed query plans?

Decoding Byte Arrays Back to UnsafeRows — `decodeUnsafeRows` Method

Caution`FIXME`

Compressing Partitions of UnsafeRows (to Byte Arrays) After Executing Physical Operator — `getByteArrayRdd` Internal Method

```
getByteArrayRdd(n: Int = -1): RDD[Array[Byte]]
```

Caution`FIXME`

resetMetrics Method

```
resetMetrics(): Unit
```

`resetMetrics` takes metrics and request them to [reset](#).

Note	<code>resetMetrics</code> is used when...FIXME
------	--

prepareSubqueries Method

Caution	FIXME
---------	-------

executeToIterator Method

Caution	FIXME
---------	-------

executeCollectIterator Method

`executeCollectIterator(): (Long, Iterator[InternalRow])`

`executeCollectIterator` ...FIXME

Note	<code>executeCollectIterator</code> is used when...FIXME
------	--

Preparing SparkPlan for Query Execution

— executeQuery Final Method

`executeQuery[T](query: => T): T`

`executeQuery` executes the input `query` in a named scope (i.e. so that all RDDs created will have the same scope for visualization like web UI).

Internally, `executeQuery` calls [prepare](#) and [waitForSubqueries](#) followed by executing `query`.

Note	<code>executeQuery</code> is executed as part of execute , executeBroadcast and when <code>CodegenSupport</code> -enabled physical operator produces a Java source code .
------	---

Broadcasting Result of Structured Query

— executeBroadcast Final Method

`executeBroadcast[T](): broadcast.Broadcast[T]`

`executeBroadcast` returns the result of a structured query as a broadcast variable.

Internally, `executeBroadcast` calls `doExecuteBroadcast` inside `executeQuery`.

Note

`executeBroadcast` is called in `BroadcastHashJoinExec`, `BroadcastNestedLoopJoinExec` and `ReusedExchangeExec` physical operators.

Performance Metrics — `metrics` Method

```
metrics: Map[String, SQLMetric] = Map.empty
```

`metrics` returns the `SQLMetrics` by their names.

By default, `metrics` contains no `SQLMetrics` (i.e. `Map.empty`).

Note

`metrics` is used when...FIXME

Taking First N UnsafeRows — `executeTake` Method

```
executeTake(n: Int): Array[InternalRow]
```

`executeTake` gives an array of up to `n` first `internal rows`.

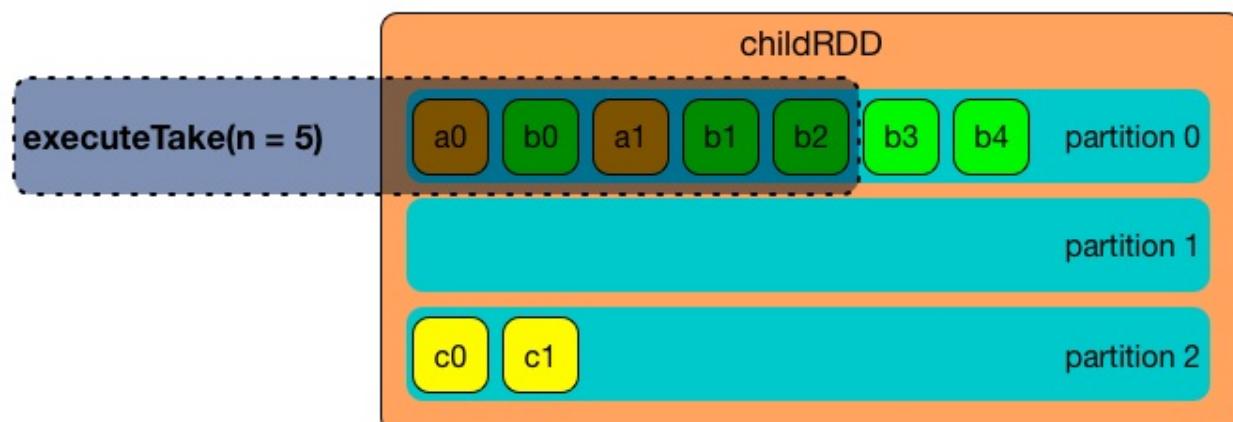


Figure 4. SparkPlan's `executeTake` takes 5 elements

Internally, `executeTake` gets an RDD of byte array of `n` unsafe rows and scans the RDD partitions one by one until `n` is reached or all partitions were processed.

`executeTake` runs Spark jobs that take all the elements from requested number of partitions, starting from the 0th partition and increasing their number by `spark.sql.limit.scaleUpFactor` property (but minimum twice as many).

Note

`executeTake` uses `SparkContext.runJob` to run a Spark job.

In the end, `executeTake` decodes the unsafe rows.

Note	<code>executeTake</code> gives an empty collection when <code>n</code> is 0 (and no Spark job is executed).
Note	<code>executeTake</code> may take and decode more unsafe rows than really needed since all unsafe rows from a partition are read (if the partition is included in the scan).

```

import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, 10)

// 8 groups over 10 partitions
// only 7 partitions are with numbers
val nums = spark.
  range(start = 0, end = 20, step = 1, numPartitions = 4).
  repartition($"id" % 8)

import scala.collection.Iterator
val showElements = (it: Iterator[java.lang.Long]) => {
  val ns = it.toSeq
  import org.apache.spark.TaskContext
  val pid = TaskContext.get.partitionId
  println(s"[partition: $pid][size: ${ns.size}] ${ns.mkString(" ")}")
}

// ordered by partition id manually for demo purposes
scala> nums.foreachPartition(showElements)
[partition: 0][size: 2] 4 12
[partition: 1][size: 2] 7 15
[partition: 2][size: 0]
[partition: 3][size: 0]
[partition: 4][size: 0]
[partition: 5][size: 5] 0 6 8 14 16
[partition: 6][size: 0]
[partition: 7][size: 3] 3 11 19
[partition: 8][size: 5] 2 5 10 13 18
[partition: 9][size: 3] 1 9 17

scala> println(spark.sessionState.conf.limitScaleUpFactor)
4

// Think how many Spark jobs will the following queries run?
// Answers follow
scala> nums.take(13)
res0: Array[Long] = Array(4, 12, 7, 15, 0, 6, 8, 14, 16, 3, 11, 19, 2)

// The number of Spark jobs = 3

scala> nums.take(5)
res34: Array[Long] = Array(4, 12, 7, 15, 0)

// The number of Spark jobs = 4

scala> nums.take(3)
res38: Array[Long] = Array(4, 12, 7)

// The number of Spark jobs = 2

```

Note	<p><code>executeTake</code> is used when:</p> <ul style="list-style-type: none"> • <code>CollectLimitExec</code> is requested to <code>executeCollect</code> • <code>AnalyzeColumnCommand</code> is executed
------	--

Executing Physical Operator and Collecting Results — `executeCollect` Method

```
executeCollect(): Array[InternalRow]
```

`executeCollect` executes the physical operator and compresses partitions of `UnsafeRows` as byte arrays (that yields a `RDD[(Long, Array[Byte])]` and so no real Spark jobs may have been submitted).

`executeCollect` runs a Spark job to `collect` the elements of the RDD and for every pair in the result (of a count and bytes per partition) decodes the byte arrays back to `UnsafeRows` and stores the decoded arrays together as the final `Array[InternalRow]`.

Note	<p><code>executeCollect</code> runs a Spark job using Spark Core's <code>RDD.collect</code> operator.</p>
Note	<p><code>executeCollect</code> returns <code>Array[InternalRow]</code>, i.e. keeps the internal representation of rows unchanged and does not convert rows to JVM types.</p>

Note	<p><code>executeCollect</code> is used when:</p> <ul style="list-style-type: none"> • <code>Dataset</code> is requested for the logical plan (being a single <code>Command</code> or their <code>Union</code>) • <code>explain</code> and <code>count</code> operators are executed • <code>Dataset</code> is requested to <code>collectFromPlan</code> • <code>SubqueryExec</code> is requested to prepare for execution (and initializes <code>relationFuture</code> for the first time) • <code>SparkPlan</code> is requested to <code>executeCollectPublic</code> • <code>ScalarSubquery</code> and <code>InSubquery</code> plan expressions are requested to <code>updateResult</code>
------	---

executeCollectPublic Method

```
executeCollectPublic(): Array[Row]
```

```
executeCollectPublic ...FIXME
```

Note

`executeCollectPublic` is used when...FIXME

newPredicate Method

```
newPredicate(expression: Expression, inputSchema: Seq[Attribute]): GenPredicate
```

```
newPredicate ...FIXME
```

Note

`newPredicate` is used when...FIXME

Waiting for Subqueries to Finish — waitForSubqueries Method

```
waitForSubqueries(): Unit
```

`waitForSubqueries` requests every [ExecSubqueryExpression](#) in [runningSubqueries](#) to updateResult.

Note

`waitForSubqueries` is used exclusively when a physical operator is requested to [prepare itself for query execution](#) (when it is [executed](#) or requested to [executeBroadcast](#)).

Output Data Partitioning Requirements — outputPartitioning Method

```
outputPartitioning: Partitioning
```

`outputPartitioning` specifies the **output data partitioning requirements**, i.e. a hint for the Spark Physical Optimizer for the number of partitions the output of the physical operator should be split across.

`outputPartitioning` defaults to a `UnknownPartitioning` (with 0 partitions).

Note	<p><code>outputPartitioning</code> is used when:</p> <ul style="list-style-type: none"> • EnsureRequirements physical query optimization is executed (and in particular adds an ExchangeCoordinator for adaptive query execution, enforces partition requirements and reorderJoinPredicates) • <code>Dataset</code> is requested to checkpoint
------	--

Output Data Ordering Requirements — `outputOrdering` Method

```
outputOrdering: Seq[SortOrder]
```

`outputOrdering` specifies the **output data ordering requirements** of the physical operator, i.e. a hint for the Spark Physical Optimizer for the sorting (ordering) of the data (within and across partitions).

`outputOrdering` defaults to no ordering (`Nil`).

Note	<p><code>outputOrdering</code> is used when:</p> <ul style="list-style-type: none"> • EnsureRequirements physical query optimization is executed (and enforces partition requirements) • <code>Dataset</code> is requested to checkpoint • <code>FileFormatWriter</code> is requested to write a query result
------	--

CodegenSupport Contract — Physical Operators with Java Code Generation

`CodegenSupport` is the [contract](#) of [physical operators](#) that want to support [Java code generation](#) and participate in the [Whole-Stage Java Code Generation \(Whole-Stage CodeGen\)](#).

```
package org.apache.spark.sql.execution

trait CodegenSupport extends SparkPlan {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def doProduce(ctx: CodegenContext): String
    def inputRDDs(): Seq[RDD[InternalRow]]

    // ...except the following that throws an UnsupportedOperationException by default
    def doConsume(ctx: CodegenContext, input: Seq[ExprCode], row: ExprCode): String
}
```

Table 1. (Subset of) CodegenSupport Contract

Method	Description		
<code>doConsume</code>	<p>Generating a plain Java source code for whole-stage "consume" path code generation</p> <p>Used exclusively when <code>CodegenSupport</code> is requested for the Java source code to consume the generated columns or a row from a physical operator.</p>		
<code>doProduce</code>	<p>Generating a plain Java source code (as a text) for the "produce" path in whole-stage Java code generation.</p> <p>Used exclusively when a physical operator is requested to generate the Java source code for produce code path, i.e. a Java code that reads the rows from the input RDDs, processes them to produce output rows that are then the input rows to downstream physical operators.</p>		
<code>inputRDDs</code>	<p>Input RDDs of a physical operator</p> <table border="1"> <tr> <td>Note</td><td>Up to two input RDDs are supported only.</td></tr> </table> <p>Used when <code>WholeStageCodegenExec</code> unary physical operator is executed.</p>	Note	Up to two input RDDs are supported only.
Note	Up to two input RDDs are supported only.		

`CodegenSupport` has the [final methods](#) that are used to generate the Java source code in different phases of Whole-Stage Java Code Generation.

Table 2. SparkPlan's Final Methods

Name	Description
<code>consume</code>	Code for consuming generated columns or a row from a physical operator <code>consume(ctx: CodegenContext, outputVars: Seq[ExprCode], row: String = null)</code>
<code>produce</code>	Code for "produce" code path <code>produce(ctx: CodegenContext, parent: CodegenSupport): String</code>

`CodegenSupport` allows physical operators to disable Java code generation.

Tip	Use <code>debugCodegen</code> or <code>QueryExecution.debug.codegen</code> methods to access the generated Java source code for a structured query.
-----	---

`variablePrefix` is...FIXME

`CodegenSupport` uses a `parent` physical operator (with `CodegenSupport`) for...FIXME

```
val q = spark.range(1)

import org.apache.spark.sql.execution.debug._
scala> q.debugCodegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Range (0, 1, step=1, splits=8)

Generated code:
...
// The above is equivalent to the following method chain
scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Range (0, 1, step=1, splits=8)

Generated code:
...
```

Table 3. CodegenSupports

CodegenSupport	Description
BaseLimitExec	
BroadcastHashJoinExec	
ColumnarBatchScan	
DataSourceScanExec	
DebugExec	
DeserializeToObjectExec	
ExpandExec	
FilterExec	
GenerateExec	
HashAggregateExec	
InputAdapter	
MapElementsExec	
ProjectExec	
RangeExec	
SampleExec	
SerializeFromObjectExec	
SortExec	
SortMergeJoinExec	
WholeStageCodegenExec	

supportCodegen Flag

```
supportCodegen: Boolean = true
```

`supportCodegen` flag is to select between `InputAdapter` or `WholeStageCodegenExec` physical operators when `CollapseCodegenStages` is executed (and checks whether a physical operator meets the requirements of whole-stage Java code generation or not).

`supportCodegen` flag is turned on by default.

Note

`supportCodegen` is turned off in the following physical operators:

- `GenerateExec`
- `HashAggregateExec` with `ImperativeAggregates`
- `SortMergeJoinExec` for all join types except `INNER` and `CROSS`

Generating Java Source Code for Produce Code Path — `produce` Final Method

```
produce(ctx: CodegenContext, parent: CodegenSupport): String
```

`produce` generates the Java source code for whole-stage-codegen produce code path for processing the rows from the input RDDs, i.e. a Java code that reads the rows from the input RDDs, processes them to produce output rows that are then the input rows to downstream physical operators.

Internally, `produce` prepares a physical operator for query execution and then generates a Java source code with the result of `doProduce`.

While generating the Java source code, `produce` annotates code blocks with `PRODUCE` markers that are simple descriptions of the physical operators in a structured query.

Tip

Enable `spark.sql.codegen.comments` Spark SQL property to have `PRODUCE` markers in the generated Java source code.

```
// ./bin/spark-shell --conf spark.sql.codegen.comments=true
import org.apache.spark.sql.execution.debug._
val q = Seq((0 to 4).toList).toDF.
  select(explode('value) as "id").
  join(spark.range(1), "id")
scala> q.debugCodegen
Found 2 WholeStageCodegen subtrees.
== Subtree 1 / 2 ==
*Range (0, 1, step=1, splits=8)
...
/* 080 */ protected void processNext() throws java.io.IOException {
/* 081 */     // PRODUCE: Range (0, 1, step=1, splits=8)
/* 082 */     // initialize Range
/* 083 */     if (!range_initRange) {
...
== Subtree 2 / 2 ==
*Project [id#6]
+- *BroadcastHashJoin [cast(id#6 as bigint)], [id#9L], Inner, BuildRight
  :- Generate explode(value#1), false, false, [id#6]
    : +- LocalTableScan [value#1]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
    +- *Range (0, 1, step=1, splits=8)
...
/* 062 */ protected void processNext() throws java.io.IOException {
/* 063 */     // PRODUCE: Project [id#6]
/* 064 */     // PRODUCE: BroadcastHashJoin [cast(id#6 as bigint)], [id#9L], Inner, Bu
ildRight
/* 065 */     // PRODUCE: InputAdapter
/* 066 */     while (inputadapter_input.hasNext() && !stopEarly()) {
...

```

Note

`produce` is used when:

- (most importantly) `WholeStageCodegenExec` is requested to generate the [Java source code for a child physical plan subtree](#) (i.e. a physical operator and its children)
- A physical operator (with `CodegenSupport`) is requested to generate a [Java source code for the produce path in whole-stage Java code generation](#) that usually looks as follows:

```
protected override def doProduce(ctx: CodegenContext): String = {
  child.asInstanceOf[CodegenSupport].produce(ctx, this)
}
```

prepareRowVar Internal Method

```
prepareRowVar(ctx: CodegenContext, row: String, colVars: Seq[ExprCode]): ExprCode
```

`prepareRowVar ...FIXME`

Note	<code>prepareRowVar</code> is used exclusively when <code>codegenSupport</code> is requested to <code>consume</code> (and <code>constructDoConsumeFunction</code> with <code>spark.sql.codegen.splitConsumeFuncByOperator</code> enabled).
------	--

constructDoConsumeFunction Internal Method

```
constructDoConsumeFunction(
  ctx: CodegenContext,
  inputVars: Seq[ExprCode],
  row: String): String
```

`constructDoConsumeFunction ...FIXME`

Note	<code>constructDoConsumeFunction</code> is used exclusively when <code>CodeGenSupport</code> is requested to <code>consume</code> .
------	---

registerComment Method

```
registerComment(text: => String): String
```

`registerComment ...FIXME`

Note	<code>registerComment</code> is used when...FIXME
------	---

metricTerm Method

```
metricTerm(ctx: CodegenContext, name: String): String
```

`metricTerm ...FIXME`

Note	<code>metricTerm</code> is used when...FIXME
------	--

usedInputs Method

```
usedInputs: AttributeSet
```

`usedInputs` returns the [expression references](#).

Note Physical operators can mark it as empty to defer evaluation of attribute expressions until they are actually used (in the [generated Java source code for consume path](#)).

Note `usedInputs` is used exclusively when `CodegenSupport` is requested to [generate a Java source code for consume path](#).

Generating Java Source Code to Consume Generated Columns or Row From Current Physical Operator

— `consume` Final Method

```
consume(ctx: CodegenContext, outputVars: Seq[ExprCode], row: String = null): String
```

Note `consume` is a final method that cannot be changed and is the foundation of codegen support.

`consume` creates the `ExprCodes` for the input variables (aka `inputVars`).

- If `outputVars` is defined, `consume` makes sure that their number is exactly the length of the `output` and copies them. In other words, `inputVars` is exactly `outputVars`.
- If `outputVars` is not defined, `consume` makes sure that `row` is defined. `consume` sets `currentVars` of the `CodegenContext` to `null` while `INPUT_ROW` to the `row`. For every attribute in the `output`, `consume` creates a `BoundReference` and requests it to [generate code for expression evaluation](#).

`consume` [creates a row variable](#).

`consume` [sets the following in the](#) `CodegenContext`:

- `currentVars` as the `inputVars`
- `INPUT_ROW` as `null`
- `freshNamePrefix` as the `variablePrefix` of the [parent CodegenSupport operator](#).

`consume` [evaluateRequiredVariables](#) (with the `output`, `inputVars` and `usedInputs` of the [parent CodegenSupport operator](#)) and creates so-called `evaluated`.

`consume` creates a so-called `consumeFunc` by `constructDoConsumeFunction` when the following are all met:

1. `spark.sqlcodegen.splitConsumeFuncByOperator` internal configuration property is enabled

2. `usedInputs` of the [parent CodegenSupport operator](#) contains all [output attributes](#)
3. `paramLength` is correct ([FIXME](#))

Otherwise, `consume` requests the [parent CodegenSupport operator](#) to [doConsume](#).

In the end, `consume` gives the plain Java source code with the comment `CONSUME`:

[parent] :

```
[evaluated]
[consumeFunc]
```

Tip

Enable [spark.sql.codegen.comments](#) Spark SQL property to have `CONSUME` markers in the generated Java source code.

```
// ./bin/spark-shell --conf spark.sql.codegen.comments=true
import org.apache.spark.sql.execution.debug._
val q = Seq((0 to 4).toList).toDF.
  select(explode('value) as "id").
  join(spark.range(1), "id")
scala> q.debugCodegen
Found 2 WholeStageCodegen subtrees.
...
== Subtree 2 / 2 ==
*Project [id#6]
+- *BroadcastHashJoin [cast(id#6 as bigint)], [id#9L], Inner, BuildRight
  :- Generate explode(value#1), false, false, [id#6]
    :  +- LocalTableScan [value#1]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
    +- *Range (0, 1, step=1, splits=8)
...
/* 066 */      while (inputadapter_input.hasNext() && !stopEarly()) {
/* 067 */          InternalRow inputadapter_row = (InternalRow) inputadapter_input.next()
;
/* 068 */          // CONSUME: BroadcastHashJoin [cast(id#6 as bigint)], [id#9L], Inner,
BuildRight
/* 069 */          // input[0, int, false]
/* 070 */          int inputadapter_value = inputadapter_row.getInt(0);
...
/* 079 */          // find matches from HashedRelation
/* 080 */          UnsafeRow bhj_matched = bhj_isNull ? null: (UnsafeRow)bhj_relation.get
Value(bhj_value);
/* 081 */          if (bhj_matched != null) {
/* 082 */              {
/* 083 */                  bhj_numOutputRows.add(1);
/* 084 */              }
/* 085 */              // CONSUME: Project [id#6]
/* 086 */              // CONSUME: WholeStageCodegen
/* 087 */              project_rowWriter.write(0, inputadapter_value);
/* 088 */              append(project_result);
/* 089 */          }
/* 090 */      }
/* 091 */  }
/* 092 */  if (shouldStop()) return;
...
```

Note	<p><code>consume</code> is used when:</p> <ul style="list-style-type: none"> • <code>BroadcastHashJoinExec</code>, <code>BaseLimitExec</code>, <code>DeserializeToObjectExec</code>, <code>ExpandExec</code>, <code>FilterExec</code>, <code>GenerateExec</code>, <code>ProjectExec</code>, <code>SampleExec</code>, <code>SerializeFromObjectExec</code>, <code>MapElementsExec</code>, <code>DebugExec</code> physical operators are requested to generate the Java source code for "consume" path in whole-stage code generation • <code>ColumnarBatchScan</code>, <code>HashAggregateExec</code>, <code>InputAdapter</code>, <code>RowDataSourceScanExec</code>, <code>RangeExec</code>, <code>SortExec</code>, <code>SortMergeJoinExec</code> physical operators are requested to generate the Java source code for the "produce" path in whole-stage code generation
------	--

parent Internal Variable Property

`parent: CodegenSupport`

`parent` is a physical operator that supports whole-stage Java code generation.

`parent` starts empty, (i.e. defaults to `null` value) and is assigned a physical operator (with `CodegenContext`) only when `CodegenContext` is requested to generate a Java source code for produce code path. The physical operator is passed in as an input argument for the produce code path.

Note

`parent` is used when...FIXME

DataSourceScanExec Contract — Leaf Physical Operators to Scan Over BaseRelation

`DataSourceScanExec` is the [contract](#) of [leaf physical operators](#) that represent scans over [BaseRelation](#).

Note	There are two <code>DataSourceScanExecs</code> , i.e. <code>FileSourceScanExec</code> and <code>RowDataSourceScanExec</code> , with a scan over data in <code>HadoopFsRelation</code> and generic <code>BaseRelation</code> relations, respectively.
------	--

`DataSourceScanExec` supports [Java code generation](#) (aka `codegen`)

```
package org.apache.spark.sql.execution

trait DataSourceScanExec extends LeafExecNode with CodegenSupport {
    // only required vals and methods that have no implementation
    // the others follow
    def metadata: Map[String, String]
    val relation: BaseRelation
    val tableIdentifier: Option[TableIdentifier]
}
```

Table 1. (Subset of) `DataSourceScanExec` Contract

Property	Description
<code>metadata</code>	Metadata (as a collection of key-value pairs) that describes the scan when requested for the simple text representation .
<code>relation</code>	<code>BaseRelation</code> that is used in the node name and...FIXME
<code>tableIdentifier</code>	Optional <code>TableIdentifier</code>

Note	The prefix for variable names for <code>DataSourceScanExec</code> operators in a generated Java source code is <code>scan</code> .
------	--

The default **node name prefix** is an empty string (that is used in the [simple node description](#)).

`DataSourceScanExec` uses the `BaseRelation` and the `TableIdentifier` as the **node name** in the following format:

```
Scan [relation] [tableIdentifier]
```

Table 2. DataSourceScanExecs

DataSourceScanExec	Description
FileSourceScanExec	
RowDataSourceScanExec	

Simple (Basic) Text Node Description (in Query Plan Tree) — simpleString Method

`simpleString: String`

Note `simpleString` is part of [QueryPlan Contract](#) to give the simple text description of a `TreeNode` in a query plan tree.

`simpleString` creates a text representation of every key-value entry in the `metadata...`

FIXME

Internally, `simpleString` sorts the `metadata` and concatenate the keys and the values (separated by the `: ``). While doing so, `simpleString` [redacts sensitive information](#) in every value and abbreviates it to the first 100 characters.

`simpleString` uses Spark Core's `utils` to `truncatedString`.

In the end, `simpleString` returns a text representation that is made up of the `nodeNamePrefix`, the `nodeName`, the `output` (schema attributes) and the `metadata` and is of the following format:

`[nodeNamePrefix][nodeName][[output]][metadata]`

```

val scanExec = basicDataSourceScanExec
scala> println(scanExec.simpleString)
Scan $line143.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$anon$1@57d94b26 [] PushedFilters:
[], ReadSchema: struct<>

def basicDataSourceScanExec = {
  import org.apache.spark.sql.catalyst.expressions.AttributeReference
  val output = Seq.empty[AttributeReference]
  val requiredColumnsIndex = output.indices
  import org.apache.spark.sql.sources.Filter
  val filters, handledFilters = Set.empty[Filter]
  import org.apache.spark.sql.catalyst.InternalRow
  import org.apache.spark.sql.catalyst.expressions.UnsafeRow
  val row: InternalRow = new UnsafeRow(0)
  val rdd: RDD[InternalRow] = sc.parallelize(row :: Nil)

  import org.apache.spark.sql.sources.{BaseRelation, TableScan}
  val baseRelation: BaseRelation = new BaseRelation with TableScan {
    import org.apache.spark.SQLContext
    val sqlContext: SQLContext = spark.sqlContext

    import org.apache.spark.sql.types.StructType
    val schema: StructType = new StructType()

    import org.apache.spark.rdd.RDD
    import org.apache.spark.sql.Row
    def buildScan(): RDD[Row] = ???
  }

  val tableIdentifier = None
  import org.apache.spark.sql.execution.RowDataSourceScanExec
  RowDataSourceScanExec(
    output, requiredColumnsIndex, filters, handledFilters, rdd, baseRelation, tableIdentifier)
}

```

verboseString Method

verboseString: String

Note	verboseString is part of QueryPlan Contract to...FIXME.
-------------	---

verboseString simply returns the redacted sensitive information in verboseString (of the parent `QueryPlan`).

Text Representation of All Nodes in Tree— treeString Method

```
treeString(verbose: Boolean, addSuffix: Boolean): String
```

Note	treeString is part of TreeNode Contract to...FIXME.
------	---

treeString simply returns the redacted sensitive information in the text representation of all nodes (in query plan tree) (of the parent TreeNode).

Redacting Sensitive Information — `redact` Internal Method

```
redact(text: String): String
```

redact ...FIXME

Note	redact is used when DataSourceScanExec is requested for the simple, verbose and tree text representations.
------	--

ColumnarBatchScan Contract — Physical Operators With Vectorized Reader

`ColumnarBatchScan` is an [extension](#) of [CodegenSupport](#) contract for physical operators that support columnar batch scan (aka **vectorized reader**).

`ColumnarBatchScan` uses the `supportsBatch` flag that is enabled (i.e. `true`) by default. It is expected that physical operators would override it to support vectorized decoding only when specific conditions are met (i.e. [FileSourceScanExec](#), [InMemoryTableScanExec](#) and [DataSourceV2ScanExec](#) physical operators).

`ColumnarBatchScan` uses the `needsUnsafeRowConversion` flag to control the name of the variable for an input row while generating the Java source code to consume generated columns or row from a physical operator that is used while generating the Java source code for producing rows. `needsUnsafeRowConversion` flag is enabled (i.e. `true`) by default that gives no name for the row term.

Table 1. ColumnarBatchScan's Performance Metrics

Key	Name (in web UI)	Description
<code>numOutputRows</code>	number of output rows	
<code>scanTime</code>	scan time	

Table 2. ColumnarBatchScans

ColumnarBatchScan	Description
DataSourceV2ScanExec	Supports vectorized decoding for SupportsScanColumnarBatch data readers that can read data in batch (default: <code>true</code>)
FileSourceScanExec	Supports vectorized decoding for FileFormats that support returning columnar batches (default: <code>false</code>)
InMemoryTableScanExec	Supports vectorized decoding when all of the following hold: <ul style="list-style-type: none"> <code>spark.sql.inMemoryColumnarStorage.enableVectorizedRead</code> property is enabled (default: <code>true</code>) Uses primitive data types only for the output schema Number of fields in the output schema is not more than <code>spark.sql.codegen.maxFields</code> property (default: <code>100</code>)

genCodeColumnVector Internal Method

```
genCodeColumnVector(
    ctx: CodegenContext,
    columnVar: String,
    ordinal: String,
    dataType: DataType,
    nullable: Boolean): ExprCode
```

`genCodeColumnVector ...FIXME`

Note

`genCodeColumnVector` is used exclusively when `ColumnarBatchScan` is requested to produceBatches.

Generating Java Source Code to Produce Columnar Batches (for Vectorized Reading)— `produceBatches` Internal Method

```
produceBatches(ctx: CodegenContext, input: String): String
```

`produceBatches` gives the Java source code to produce batches...FIXME

```
// Example to show produceBatches to generate a Java source code
// Uses InMemoryTableScanExec as a ColumnarBatchScan

// Create a DataFrame
val ids = spark.range(10)
// Cache it (and trigger the caching since it is lazy)
ids.cache.foreach(_ => ())

import org.apache.spark.sql.execution.columnar.InMemoryTableScanExec
// we need executedPlan with WholeStageCodegenExec physical operator
// this will make sure the code generation starts at the right place
val plan = ids.queryExecution.executedPlan
val scan = plan.collectFirst { case e: InMemoryTableScanExec => e }.get

assert(scan.supportsBatch, "supportsBatch flag should be on to trigger produceBatches"
)

import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext

// produceBatches is private so we have to trigger it from "outside"
// It could be doProduce with supportsBatch flag on but it is protected
// (doProduce will also take care of the extra input `input` parameter)
// let's do this the only one right way
import org.apache.spark.sql.execution.CodegenSupport
val parent = plan.p(0).asInstanceOf[CodegenSupport]
val produceCode = scan.produce(ctx, parent)
```

```

scala> println(produceCode)

if (inmemorytablescan.mutableStateArray1[1] == null) {
    inmemorytablescan_nextBatch1();
}
while (inmemorytablescan.mutableStateArray1[1] != null) {
    int inmemorytablescan_numRows1 = inmemorytablescan.mutableStateArray1[1].numRows();
    int inmemorytablescan_localEnd1 = inmemorytablescan_numRows1 - inmemorytablescan_batchIdx1;
    for (int inmemorytablescan_localIdx1 = 0; inmemorytablescan_localIdx1 < inmemorytablescan_localEnd1; inmemorytablescan_localIdx1++) {
        int inmemorytablescan_rowIdx1 = inmemorytablescan_batchIdx1 + inmemorytablescan_localIdx1;
        long inmemorytablescan_value2 = inmemorytablescan.mutableStateArray2[1].getLong(inmemorytablescan_rowIdx1);
        inmemorytablescan.mutableStateArray5[1].write(0, inmemorytablescan_value2);
        append(inmemorytablescan.mutableStateArray3[1]);
        if (shouldStop()) { inmemorytablescan_batchIdx1 = inmemorytablescan_rowIdx1 + 1; return; }
    }
    inmemorytablescan_batchIdx1 = inmemorytablescan_numRows1;
    inmemorytablescan.mutableStateArray1[1] = null;
    inmemorytablescan_nextBatch1();
}
((org.apache.spark.sql.execution.metric.SQLMetric) references[3] /* scanTime */).add(inmemorytablescan_scanTime1 / (1000 * 1000));
inmemorytablescan_scanTime1 = 0;

// the code does not look good and begs for some polishing
// (You can only imagine how the Polish me looks when I say "polishing" :))

import org.apache.spark.sql.execution.WholeStageCodegenExec
val wsce = plan.asInstanceOf[WholeStageCodegenExec]

// Trigger code generation of the entire query plan tree
val (ctx, code) = wsce.doCodeGen

// CodeFormatter can pretty-print the code
import org.apache.spark.sql.catalyst.expressions.codegen.CodeFormatter
println(CodeFormatter.format(code))

```

Note

`produceBatches` is used exclusively when `ColumnarBatchScan` is requested to generate the Java source code for produce path in whole-stage code generation (when `supportsBatch` flag is on).

supportsBatch Method

```
supportsBatch: Boolean = true
```

`supportsBatch` flag controls whether a [FileFormat](#) supports [vectorized decoding](#) or not. `supportsBatch` is enabled (i.e. `true`) by default.

Note

- `supportsBatch` is used when:
 - `ColumnarBatchScan` is requested to generate the Java source code for [produce path in whole-stage code generation](#)
 - `FileSourceScanExec` physical operator is requested for [metadata](#) (for **Batched** metadata) and to [execute](#)
 - `InMemoryTableScanExec` physical operator is requested for [supportCodegen](#) flag, [input RDD](#) and to [execute](#)
 - `DataSourceV2ScanExec` physical operator is requested to [execute](#)

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

```
doProduce(ctx: CodegenContext): String
```

Note

`doProduce` is part of [CodegenSupport Contract](#) to generate the Java source code for [produce path](#) in Whole-Stage Code Generation.

`doProduce` firstly requests the input `CodegenContext` to [add a mutable state](#) for the first input [RDD](#) of a [physical operator](#).

`doProduce` [produceBatches](#) when `supportsBatch` is enabled or [produceRows](#).

Note

`supportsBatch` is enabled by default unless overriden by a physical operator.

```
// Example 1: ColumnarBatchScan with supportsBatch enabled
// Let's create a query with a InMemoryTableScanExec physical operator that supports batch decoding
// InMemoryTableScanExec is a ColumnarBatchScan
val q = spark.range(4).cache
val plan = q.queryExecution.executedPlan

import org.apache.spark.sql.execution.columnar.InMemoryTableScanExec
val inmemoryScan = plan.collectFirst { case exec: InMemoryTableScanExec => exec }.get

assert(inmemoryScan.supportsBatch)

import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext
```

```

import org.apache.spark.sql.execution.CodegenSupport
val parent = plan.asInstanceOf[CodegenSupport]
val code = inmemoryScan.produce(ctx, parent)
scala> println(code)

if (inmemorytablescan mutableStateArray1[1] == null) {
    inmemorytablescan_nextBatch1();
}
while (inmemorytablescan mutableStateArray1[1] != null) {
    int inmemorytablescan_numRows1 = inmemorytablescan mutableStateArray1[1].numRows();
    int inmemorytablescan_localEnd1 = inmemorytablescan_numRows1 - inmemorytablescan_batchIdx1;
    for (int inmemorytablescan_localIdx1 = 0; inmemorytablescan_localIdx1 < inmemorytablescan_localEnd1; inmemorytablescan_localIdx1++) {
        int inmemorytablescan_rowIdx1 = inmemorytablescan_batchIdx1 + inmemorytablescan_localIdx1;
        long inmemorytablescan_value2 = inmemorytablescan mutableStateArray2[1].getLong(inmemorytablescan_rowIdx1);
        inmemorytablescan mutableStateArray5[1].write(0, inmemorytablescan_value2);
        append(inmemorytablescan mutableStateArray3[1]);
        if (shouldStop()) { inmemorytablescan_batchIdx1 = inmemorytablescan_rowIdx1 + 1; return; }
    }
    inmemorytablescan_batchIdx1 = inmemorytablescan_numRows1;
    inmemorytablescan mutableStateArray1[1] = null;
    inmemorytablescan_nextBatch1();
}
((org.apache.spark.sql.execution.metric.SQLMetric) references[3] /* scanTime */).add(inmemorytablescan_scanTime1 / (1000 * 1000));
inmemorytablescan_scanTime1 = 0;

// Example 2: ColumnarBatchScan with supportsBatch disabled

val q = Seq(Seq(1,2,3)).toDF("ids").cache
val plan = q.queryExecution.executedPlan

import org.apache.spark.sql.execution.columnar.InMemoryTableScanExec
val inmemoryScan = plan.collectFirst { case exec: InMemoryTableScanExec => exec }.get

assert(inmemoryScan.supportsBatch == false)

// NOTE: The following codegen won't work since supportsBatch is off and so is codegen
import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext
import org.apache.spark.sql.execution.CodegenSupport
val parent = plan.asInstanceOf[CodegenSupport]
scala> val code = inmemoryScan.produce(ctx, parent)
java.lang.UnsupportedOperationException
    at org.apache.spark.sql.execution.CodegenSupport$class.doConsume(WholeStageCodegenException.scala:315)
    at org.apache.spark.sql.execution.columnar.InMemoryTableScanExec.doConsume(InMemoryT

```

```

ableScanExec.scala:33)
  at org.apache.spark.sql.execution.CodegenSupport$class.constructDoConsumeFunction(WholeStageCodegenExec.scala:208)
  at org.apache.spark.sql.execution.CodegenSupport$class.consume(WholeStageCodegenExec.scala:179)
  at org.apache.spark.sql.execution.columnar.InMemoryTableScanExec.consume(InMemoryTableScanExec.scala:33)
  at org.apache.spark.sql.execution.ColumnarBatchScan$class.produceRows(ColumnarBatchScan.scala:166)
  at org.apache.spark.sql.execution.ColumnarBatchScan$class.doProduce(ColumnarBatchScan.scala:80)
  at org.apache.spark.sql.execution.columnar.InMemoryTableScanExec.doProduce(InMemoryTableScanExec.scala:33)
  at org.apache.spark.sql.execution.CodegenSupport$$anonfun$produce$1.apply(WholeStageCodegenExec.scala:88)
  at org.apache.spark.sql.execution.CodegenSupport$$anonfun$produce$1.apply(WholeStageCodegenExec.scala:83)
  at org.apache.spark.sql.execution.SparkPlan$$anonfun$executeQuery$1.apply(SparkPlan.scala:155)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
  at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:152)
  at org.apache.spark.sql.execution.CodegenSupport$class.produce(WholeStageCodegenExec.scala:83)
  at org.apache.spark.sql.execution.columnar.InMemoryTableScanExec.produce(InMemoryTableScanExec.scala:33)
  ...
  ... 49 elided

```

Generating Java Source Code for Producing Rows — `produceRows` Internal Method

```
produceRows(ctx: CodegenContext, input: String): String
```

`produceRows` creates a new metric term for the `numOutputRows` metric.

`produceRows` creates a fresh term name for a `row` variable and assigns it as the name of the `INPUT_ROW`.

`produceRows` resets (`nulls`) `currentVars`.

For every `output schema attribute`, `produceRows` creates a `BoundReference` and requests it to generate code for expression evaluation.

`produceRows` selects the name of the row term per `needsUnsafeRowConversion` flag.

`produceRows` generates the Java source code to consume generated columns or row from the current physical operator and uses it to generate the final Java source code for producing rows.

```
// Demo: ColumnarBatchScan.produceRows in Action
// 1. FileSourceScanExec as a ColumnarBatchScan
val q = spark.read.text("README.md")

val plan = q.queryExecution.executedPlan
import org.apache.spark.sql.execution.FileSourceScanExec
val scan = plan.collectFirst { case exec: FileSourceScanExec => exec }.get

// 2. supportsBatch is off
assert(scan.supportsBatch == false)

// 3. InMemoryTableScanExec.produce
import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext
import org.apache.spark.sql.execution.CodegenSupport

import org.apache.spark.sql.execution.WholeStageCodegenExec
val wsce = plan.collectFirst { case exec: WholeStageCodegenExec => exec }.get

val code = scan.produce(ctx, parent = wsce)
scala> println(code)
// blank lines removed
while (scan mutableStateArray[2].hasNext()) {
  InternalRow scan_row2 = (InternalRow) scan mutableStateArray[2].next();
  ((org.apache.spark.sql.execution.metric.SQLMetric) references[2] /* numOutputRows */)
).add(1);
  append(scan_row2);
  if (shouldStop()) return;
}
```

Note

`produceRows` is used exclusively when `ColumnarBatchScan` is requested to generate the Java source code for produce path in whole-stage code generation (when `supportsBatch` flag is off).

Fully-Qualified Class Names (Types) of Concrete ColumnVectors — `vectorTypes` Method

```
vectorTypes: Option[Seq[String]] = None
```

`vectorTypes` defines the fully-qualified class names (*types*) of the concrete `ColumnVectors` for every column used in a columnar batch.

`vectorTypes` gives no vector types by default (`None`).

Note

`vectorTypes` is used exclusively when `ColumnarBatchScan` is requested to `produceBatches`.

ObjectConsumerExec Contract — Unary Physical Operators with Child Physical Operator with One-Attribute Output Schema

`ObjectConsumerExec` is the [contract](#) of [unary physical operators](#) with the child physical operator using a one-attribute [output schema](#).

```
package org.apache.spark.sql.execution

trait ObjectConsumerExec extends UnaryExecNode {
    // No properties (vals and methods) that have no implementation
}
```

`ObjectConsumerExec` requests the child physical operator for the [output schema](#) attribute set when requested for the [references](#).

Table 1. ObjectConsumerExecs

ObjectConsumerExec	Description
AppendColumnsWithObjectExec	
MapElementsExec	
MapPartitionsExec	
SerializeFromObjectExec	

inputObjectType Method

```
inputObjectType: DataType
```

`inputObjectType` simply returns the [data type](#) of the single [output attribute](#) of the child physical operator.

Note	<code>inputObjectType</code> is used when...FIXME
------	---

BaseLimitExec Contract

BaseLimitExec is...FIXME

Table 1. BaseLimitExecs

BaseLimitExec	Description
GlobalLimitExec	
LocalLimitExec	

Exchange Contract — Base for Unary Physical Operators that Exchange Data

`Exchange` is the base of [unary physical operators](#) that exchange data among multiple threads or processes.

When requested for the [output schema](#), `Exchange` simply uses the child physical operator's output schema.

Table 1. Exchanges

Exchange	Description
BroadcastExchangeExec	
ShuffleExchangeExec	

Projection Contract — Functions to Produce InternalRow for InternalRow

`Projection` is a [contract](#) of Scala functions that produce an [internal binary row](#) for a given internal row.

```
Projection: InternalRow => InternalRow
```

`Projection` can optionally be [initialized](#) with the current partition index (which by default does nothing).

```
initialize(partitionIndex: Int): Unit = {}
```

Note	<code>initialize</code> is overriden by InterpretedProjection and InterpretedMutableProjection projections that are used in interpreted expression evaluation .
------	---

Table 1. Projections

Projection	Description
UnsafeProjection	
InterpretedProjection	
IdentityProjection	
MutableProjection	
InterpretedMutableProjection	<i>Appears not to be used anymore</i>

UnsafeProjection — Generic Function to Encode InternalRows to UnsafeRows

`UnsafeProjection` is a `Projection` function that encodes `InternalRows` as `UnsafeRows`.

```
UnsafeProjection: InternalRow =[apply]=> UnsafeRow
```

Spark SQL uses `UnsafeProjection` factory object to [create](#) concrete *adhoc* `UnsafeProjection` instances.

Note The base `UnsafeProjection` has no concrete named implementations and [create](#) factory methods delegate all calls to [GenerateUnsafeProjection.generate](#) in the end.

Creating `UnsafeProjection` — `create` Factory Method

```
create(schema: StructType): UnsafeProjection      (1)
create(fields: Array[DataType]): UnsafeProjection (2)
create(expr: Expression): UnsafeProjection        (3)
create(exprs: Seq[Expression], inputSchema: Seq[Attribute]): UnsafeProjection (4)
create(exprs: Seq[Expression]): UnsafeProjection   (5)
create(
  exprs: Seq[Expression],
  inputSchema: Seq[Attribute],
  subexpressionEliminationEnabled: Boolean): UnsafeProjection
```

1. `create` takes the [DataTypes](#) from `schema` and calls the 2nd `create`
2. `create` creates a [BoundReference](#) per field in `fields` and calls the 5th `create`
3. `create` calls the 5th `create`
4. `create` calls the 5th `create`
5. The main `create` that does the heavy work

`create` transforms all [CreateNamedStruct](#) expressions to `createNamedStructUnsafe` in every [BoundReference](#) in the input `exprs`.

In the end, `create` requests `GenerateUnsafeProjection` to [generate a `UnsafeProjection`](#).

Note A variant of `create` takes `subexpressionEliminationEnabled` flag (that usually is [subexpressionEliminationEnabled](#) flag of `SparkPlan`).

GenerateUnsafeProjection

`GenerateUnsafeProjection` is a [CodeGenerator](#) that generates the bytecode for a [UnsafeProjection](#) for given expressions (i.e. `CodeGenerator[Seq[Expression], UnsafeProjection]`).

```
GenerateUnsafeProjection: Seq[Expression] => UnsafeProjection
```

Tip Enable `DEBUG` logging level for `org.apache.spark.sql.catalyst.expressions.codegen.GenerateUnsafeProjection` logger happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.catalyst.expressions.codegen.GenerateUnsafePro
```

Refer to [Logging](#).

Generating `UnsafeProjection` — `generate` Method

```
generate(  
    expressions: Seq[Expression],  
    subexpressionEliminationEnabled: Boolean): UnsafeProjection
```

`generate` canonicalize the input `expressions` followed by generating a JVM bytecode for a [UnsafeProjection](#) for the expressions.

Note

- `generate` is used when:
- `UnsafeProjection` factory object is requested for a `UnsafeProjection`
 - `ExpressionEncoder` is requested to initialize the internal `UnsafeProjection`
 - `FileFormat` is requested to build a data reader with partition column values appended
 - `OrcFileFormat` is requested to build a data reader with partition column values appended
 - `ParquetFileFormat` is requested to build a data reader with partition column values appended
 - `GroupedIterator` is requested for `keyProjection`
 - `ObjectOperator` is requested to `serializeObjectToRow`
 - (Spark MLlib) `LibSVMFileFormat` is requested to `buildReader`
 - (Spark Structured Streaming) `StateStoreRestoreExec`, `StateStoreSaveExec` and `StreamingDeduplicateExec` are requested to execute

canonicalize Method

```
canonicalize(in: Seq[Expression]): Seq[Expression]
```

`canonicalize` removes unnecessary `Alias` expressions.

Internally, `canonicalize` uses `ExpressionCanonicalizer` rule executor (that in turn uses just one `CleanExpressions` expression rule).

Generating JVM Bytecode For UnsafeProjection For Given Expressions (With Optional Subexpression Elimination)

— create Method

```
create(  
  expressions: Seq[Expression],  
  subexpressionEliminationEnabled: Boolean): UnsafeProjection  
create(references: Seq[Expression]): UnsafeProjection (1)
```

1. Calls the former `create` with `subexpressionEliminationEnabled` flag off

`create` first creates a `CodegenContext` and an `Java source code` for the input expressions .

`create` creates a code body with `public java.lang.Object generate(Object[] references)` method that creates a `SpecificUnsafeProjection`.

```
public java.lang.Object generate(Object[] references) {
    return new SpecificUnsafeProjection(references);
}

class SpecificUnsafeProjection extends UnsafeProjection {
    ...
}
```

`create` creates a `CodeAndComment` with the code body and [comment placeholders](#).

You should see the following DEBUG message in the logs:

```
DEBUG GenerateUnsafeProjection: code for [expressions]:
[code]
```

Tip Enable `DEBUG` logging level for `org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator` logger to see the message above.

```
log4j.logger.org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator=DEBUG
```

See [CodeGenerator](#).

`create` requests `CodeGenerator` to compile the Java source code to JVM bytecode (using [Janino](#)).

`create` requests `CodegenContext` for [references](#) and requests the compiled class to create a `SpecificUnsafeProjection` for the input references that in the end is the final [UnsafeProjection](#).

Note (Single-argument) `create` is part of [CodeGenerator Contract](#).

Creating ExprCode for Expressions (With Optional Subexpression Elimination) — `createCode` Method

```
createCode(
  ctx: CodegenContext,
  expressions: Seq[Expression],
  useSubexprElimination: Boolean = false): ExprCode
```

`createCode` requests the input `CodegenContext` to generate a Java source code for code-generated evaluation of every expression in the input `expressions`.

`createCode ...FIXME`

```
import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext

// Use Catalyst DSL
import org.apache.spark.sql.catalyst.dsl.expressions._
val expressions = "hello".expr.as("world") :: "hello".expr.as("world") :: Nil

import org.apache.spark.sql.catalyst.expressions.codegen.GenerateUnsafeProjection
val eval = GenerateUnsafeProjection.createCode(ctx, expressions, useSubexprElimination
= true)

scala> println(eval.code)

    mutableStateArray1[0].reset();

    mutableStateArray2[0].write(0, ((UTF8String) references[0] /* literal */);

    mutableStateArray2[0].write(1, ((UTF8String) references[1] /* literal */))
;
mutableStateArray[0].setTotalSize(mutableStateArray1[0].totalSize()));

scala> println(eval.value)
mutableStateArray[0]
```

Note

`createCode` is used when:

- `CreateNamedStructUnsafe` is requested to generate a Java source code
- `GenerateUnsafeProjection` is requested to create a `UnsafeProjection`
- `CodegenSupport` is requested to `prepareRowVar` (to generate a Java source code to consume generated columns or row from a physical operator)
- `HashAggregateExec` is requested to `doProduceWithKeys` and `doConsumeWithKeys`
- `BroadcastHashJoinExec` is requested to `genStreamSideJoinKey` (when generating the Java source code for joins)

GenerateMutableProjection

GenerateMutableProjection is...FIXME

Creating MutableProjection — `create` Internal Method

```
create(  
  expressions: Seq[Expression],  
  useSubexprElimination: Boolean): MutableProjection
```

create ...FIXME

Note	create is used when...FIXME
------	-----------------------------

InterpretedProjection

InterpretedProjection is a [Projection](#) that...FIXME

InterpretedProjection takes [expressions](#) when created.

```
// HACK: Disable symbolToColumn implicit conversion
// It is imported automatically in spark-shell (and makes demos impossible)
// implicit def symbolToColumn(s: Symbol): org.apache.spark.sql.ColumnName
trait ThatWasABadIdea
implicit def symbolToColumn(ack: ThatWasABadIdea) = ack

import org.apache.spark.sql.catalyst.dsl.expressions._
val boundRef = 'hello.string.at(4)

import org.apache.spark.sql.catalyst.expressions.{Expression, Literal}
val expressions: Seq[Expression] = Seq(Literal(1), boundRef)

import org.apache.spark.sql.catalyst.expressions.InterpretedProjection
val ip = new InterpretedProjection(expressions)
scala> println(ip)
Row > [1, input[4, string, true]]
```

InterpretedProjection is [created](#) when:

- UserDefinedGenerator is requested to [initializeConverters](#)
- ConvertToLocalRelation logical optimization is executed (to transform [Project](#) logical operators)
- HiveGenericUDTF is evaluated
- ScriptTransformationExec physical operator is executed

Initializing Nondeterministic Expressions — [initialize](#) Method

```
initialize(partitionIndex: Int): Unit
```

Note	initialize is part of Projection Contract to...FIXME.
------	---

[initialize](#) requests [Nondeterministic](#) [expressions](#) (in [expressions](#)) to [initialize](#) with the [partitionIndex](#).

CodeGeneratorWithInterpretedFallback

`CodeGeneratorWithInterpretedFallback` is the base of [codegen object generators](#) that can create objects for [codegen](#) and [interpreted](#) evaluation paths.

Table 1. CodeGeneratorWithInterpretedFallback Contract

Method	Description
<code>createCodeGeneratedObject</code>	<pre>createCodeGeneratedObject(in: IN): OUT</pre> <p>Used when...FIXME</p>
<code>createInterpretedObject</code>	<pre>createInterpretedObject(in: IN): OUT</pre> <p>Used when...FIXME</p>

Note	UnsafeProjection is the one and only known implementation of the CodeGeneratorWithInterpretedFallback Contract in Apache Spark.
Note	<p><code>CodeGeneratorWithInterpretedFallback</code> is a Scala type constructor (aka <i>generic type</i>) that accepts two types referred as <code>IN</code> and <code>OUT</code>.</p> <pre>abstract class CodeGeneratorWithInterpretedFallback[IN, OUT]</pre>

createObject Method

```
createObject(in: IN): OUT
```

`createObject` ...FIXME

Note	<code>createObject</code> is used exclusively when <code>UnsafeProjection</code> is requested to create an UnsafeProjection for Catalyst expressions .
------	--

SQLMetric—SQL Execution Metric of Physical Operator

`SQLMetric` is a SQL metric for monitoring execution of a [physical operator](#).

`SQLMetric` is an [accumulator](#) (and that is the mechanism to propagate SQL metric updates on the executors to the driver, e.g. web UI).

Note	Use Details for Query page in SQL tab in web UI to see the SQL execution metrics of a structured query.
------	---

	SQL metrics are collected using <code>sparkListener</code> . If there are no tasks, Spark SQL cannot collect any metrics. Updates to metrics on the driver-side require explicit calls of SQLMetrics.postDriverMetricUpdates .
--	--

This is why executing some physical operators (e.g. `LocalTableScanExec`) may not have SQL metrics in web UI's [Details for Query](#) in SQL tab.

Note	Compare the following SQL queries and their execution pages.
------	--

```
// The query does not have SQL metrics in web UI
Seq("Jacek").toDF("name").show

// The query gives numOutputRows metric in web UI's Details for Query (SQL tab)
Seq("Jacek").toDF("name").count
```

`SQLMetric` takes a metric type and an initial value when created.

Table 1. Metric Types and Corresponding Create Methods

Metric Type	Create Method	Failed Values Counted?	Description
size	<code>createSizeMetric</code>	no	Used when...
sum	<code>createMetric</code>	no	Used when...
timing	<code>createTimingMetric</code>	no	Used when...

reset Method

```
reset(): Unit
```

```
reset ...FIXME
```

Note	<code>reset</code> is used when...FIXME
------	---

Posting Driver-Side Metric Updates

— `SQLMetrics.postDriverMetricUpdates` Method

```
postDriverMetricUpdates(  
    sc: SparkContext,  
    executionId: String,  
    metrics: Seq[SQLMetric]): Unit
```

`postDriverMetricUpdates` posts a [SparkListenerDriverAccumUpdates](#) event to [LiveListenerBus](#) when `executionId` is specified.

Note	<code>postDriverMetricUpdates</code> method belongs to <code>SQLMetrics</code> object.
------	--

Note	<p><code>postDriverMetricUpdates</code> is used when:</p> <ul style="list-style-type: none"> • <code>BroadcastExchangeExec</code> is requested to prepare for execution (and initializes relationFuture for the first time) • <code>FileSourceScanExec</code> physical operator is requested for selectedPartitions (and posts updates to <code>numFiles</code> and <code>metadataTime</code> metrics) • <code>SubqueryExec</code> physical operator is requested to prepare for execution (and initializes relationFuture for the first time that in turn posts updates to <code>collectTime</code> and <code> dataSize</code> metrics)
------	---

BroadcastExchangeExec Unary Physical Operator for Broadcast Joins

`BroadcastExchangeExec` is a [Exchange](#) unary physical operator to collect and broadcast rows of a child relation (to worker nodes).

`BroadcastExchangeExec` is [created](#) exclusively when `EnsureRequirements` physical query plan optimization [ensures BroadcastDistribution](#) of the input data of a physical operator (that can really be either [BroadcastHashJoinExec](#) or [BroadcastNestedLoopJoinExec](#) operators).

```
val t1 = spark.range(5)
val t2 = spark.range(5)
val q = t1.join(t2).where(t1("id") === t2("id"))

scala> q.explain
== Physical Plan ==
*BroadcastHashJoin [id#19L, [id#22L], Inner, BuildRight
:- *Range (0, 5, step=1, splits=Some(8))
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 5, step=1, splits=Some(8))
```

Table 1. BroadcastExchangeExec's Performance Metrics

Key	Name (in web UI)	Description
broadcastTime	time to broadcast (ms)	
buildTime	time to build (ms)	
collectTime	time to collect (ms)	
dataSize	data size (bytes)	

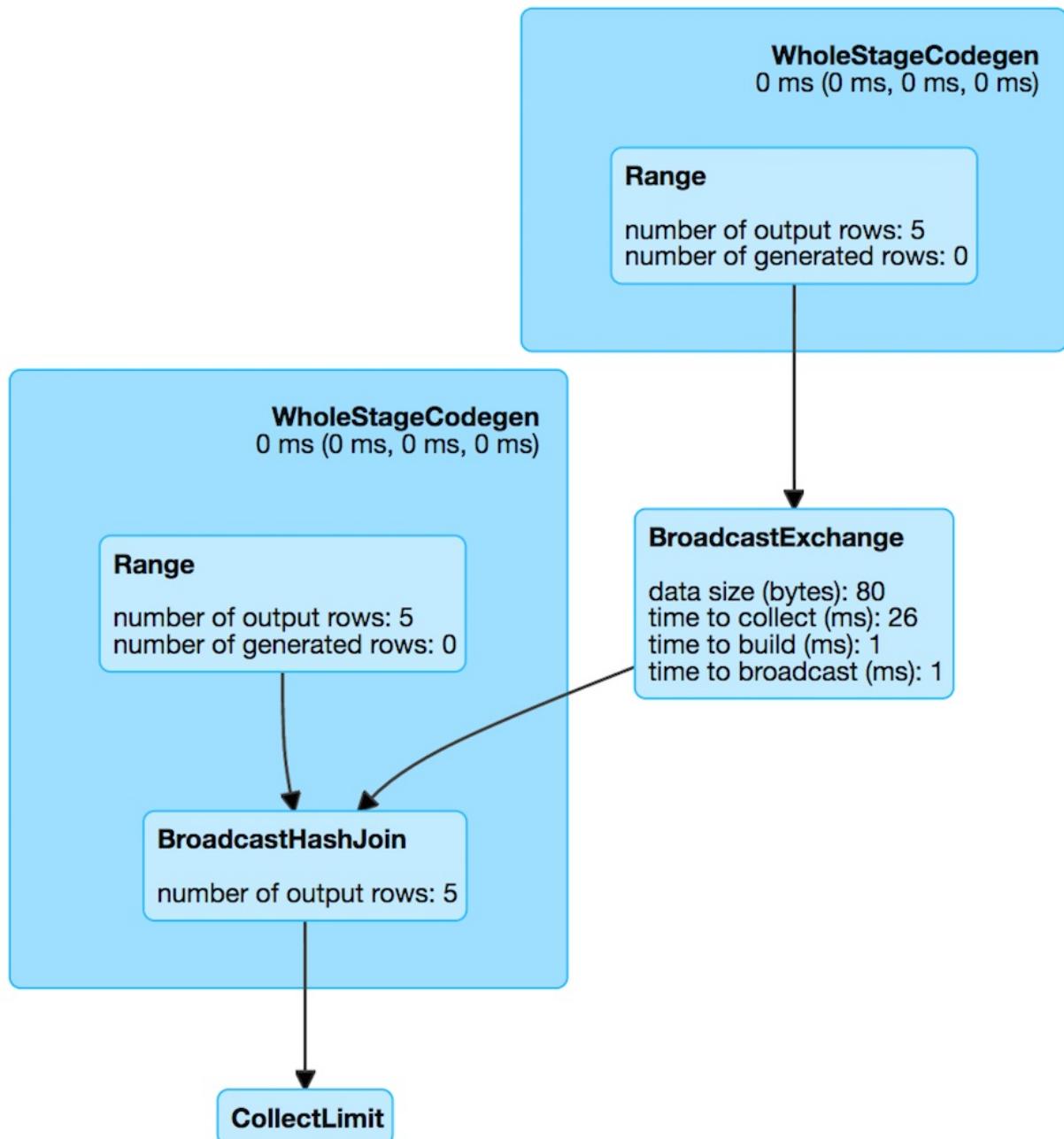


Figure 1. BroadcastExchangeExec in web UI (Details for Query)

`BroadcastExchangeExec` uses [BroadcastPartitioning](#) partitioning scheme (with the input `BroadcastMode`).

Waiting Until Relation Has Been Broadcast — `doExecuteBroadcast` Method

```
def doExecuteBroadcast[T](): broadcast.Broadcast[T]
```

`doExecuteBroadcast` waits until the [rows](#) are broadcast.

Note	<code>doExecuteBroadcast</code> waits spark.sql.broadcastTimeout (defaults to 5 minutes).
------	---

Note

`doExecuteBroadcast` is part of [SparkPlan Contract](#) to return the result of a structured query as a broadcast variable.

Lazily-Once-Initialized Asynchronously-Broadcast relationFuture Internal Attribute

```
relationFuture: Future[broadcast.Broadcast[Any]]
```

When "materialized" (aka `executed`), `relationFuture` finds the current [execution id](#) and sets it to the `Future` thread.

`relationFuture` requests [child physical operator](#) to [executeCollectIterator](#).

`relationFuture` records the time for `executeCollectIterator` in [collectTime](#) metrics.

Note

`relationFuture` accepts a relation with up to 512 millions rows and 8GB in size, and reports a `SparkException` if the conditions are violated.

`relationFuture` requests the input [BroadcastMode](#) to `transform` the internal rows to create a relation, e.g. [HashedRelation](#) or a `Array[InternalRow]`.

`relationFuture` calculates the data size:

- For a `HashedRelation`, `relationFuture` requests it to [estimatedSize](#)
- For a `Array[InternalRow]`, `relationFuture` transforms the `InternalRows` to [UnsafeRows](#) and requests each to [getSizelnBytes](#) that it sums all up.

`relationFuture` records the data size as the [dataSize](#) metric.

`relationFuture` records the [buildTime](#) metric.

`relationFuture` requests the [SparkContext](#) to `broadcast` the relation and records the time in [broadcastTime](#) metrics.

In the end, `relationFuture` requests [SQLMetrics](#) to [post a](#) [SparkListenerDriverAccumUpdates](#) (with the execution id and the SQL metrics) and returns the broadcast internal rows.

Note

Since initialization of `relationFuture` happens on the driver, [posting a](#) [SparkListenerDriverAccumUpdates](#) is the only way how all the SQL metrics could be accessible to other subsystems using `SparkListener` listeners (incl. web UI).

In case of `OutOfMemoryError`, `relationFuture` reports another `OutOfMemoryError` with the following message:

Not enough memory to build and broadcast the table to all worker nodes. As a workaround, you can either disable broadcast by setting `spark.sql.autoBroadcastJoinThreshold` to `-1` or increase the spark driver memory by setting `spark.driver.memory` to a higher value

Note	<code>relationFuture</code> is executed on a separate thread from a custom <code>scala.concurrent.ExecutionContext</code> (built from a cached <code>java.util.concurrent.ThreadPoolExecutor</code> with the prefix broadcast-exchange and up to 128 threads).
Note	<code>relationFuture</code> is used when <code>BroadcastExchangeExec</code> is requested to prepare for execution (that triggers asynchronous execution of the child operator and broadcasting the result) and execute broadcast (that waits until the broadcasting has finished).

Broadcasting Relation (Rows) Asynchronously — `doPrepare` Method

`doPrepare(): Unit`

Note	<code>doPrepare</code> is part of SparkPlan Contract to prepare a physical operator for execution.
------	--

`doPrepare` simply "materializes" the internal lazily-once-initialized [asynchronous broadcast](#).

Creating BroadcastExchangeExec Instance

`BroadcastExchangeExec` takes the following when created:

- [BroadcastMode](#)
- Child [logical plan](#)

BroadcastHashJoinExec Binary Physical Operator for Broadcast Hash Join

`BroadcastHashJoinExec` is a [binary physical operator](#) to [perform](#) a [broadcast hash join](#).

`BroadcastHashJoinExec` is [created](#) after applying [JoinSelection](#) execution planning strategy to [ExtractEquiJoinKeys](#)-destructurable logical query plans (i.e. [INNER](#), [CROSS](#), [LEFT OUTER](#), [LEFT SEMI](#), [LEFT ANTI](#)) of which the `right` physical operator [can be broadcast](#).

`BroadcastHashJoinExec` supports [Java code generation](#) (aka `codegen`).

```
val tokens = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "BroadcastHashJoinExec")
).toDF("id", "token")

scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold")
res0: String = 10485760

val q = tokens.join(tokens, Seq("id"), "inner")
scala> q.explain
== Physical Plan ==
*Project [id#15, token#16, token#21]
+- *BroadcastHashJoin [id#15], [id#20], Inner, BuildRight
  :- LocalTableScan [id#15, token#16]
  +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as
bigint)))
    +- LocalTableScan [id#20, token#21]
```

`BroadcastHashJoinExec` [requires that partition requirements](#) for the two children physical operators match [BroadcastDistribution](#) (with a [HashedRelationBroadcastMode](#)) and [UnspecifiedDistribution](#) (for [left](#) and [right](#) sides of a join or vice versa).

Table 1. `BroadcastHashJoinExec`'s Performance Metrics

Key	Name (in web UI)	Description
<code>numOutputRows</code>	number of output rows	
<code>avgHashProbe</code>	avg hash probe	

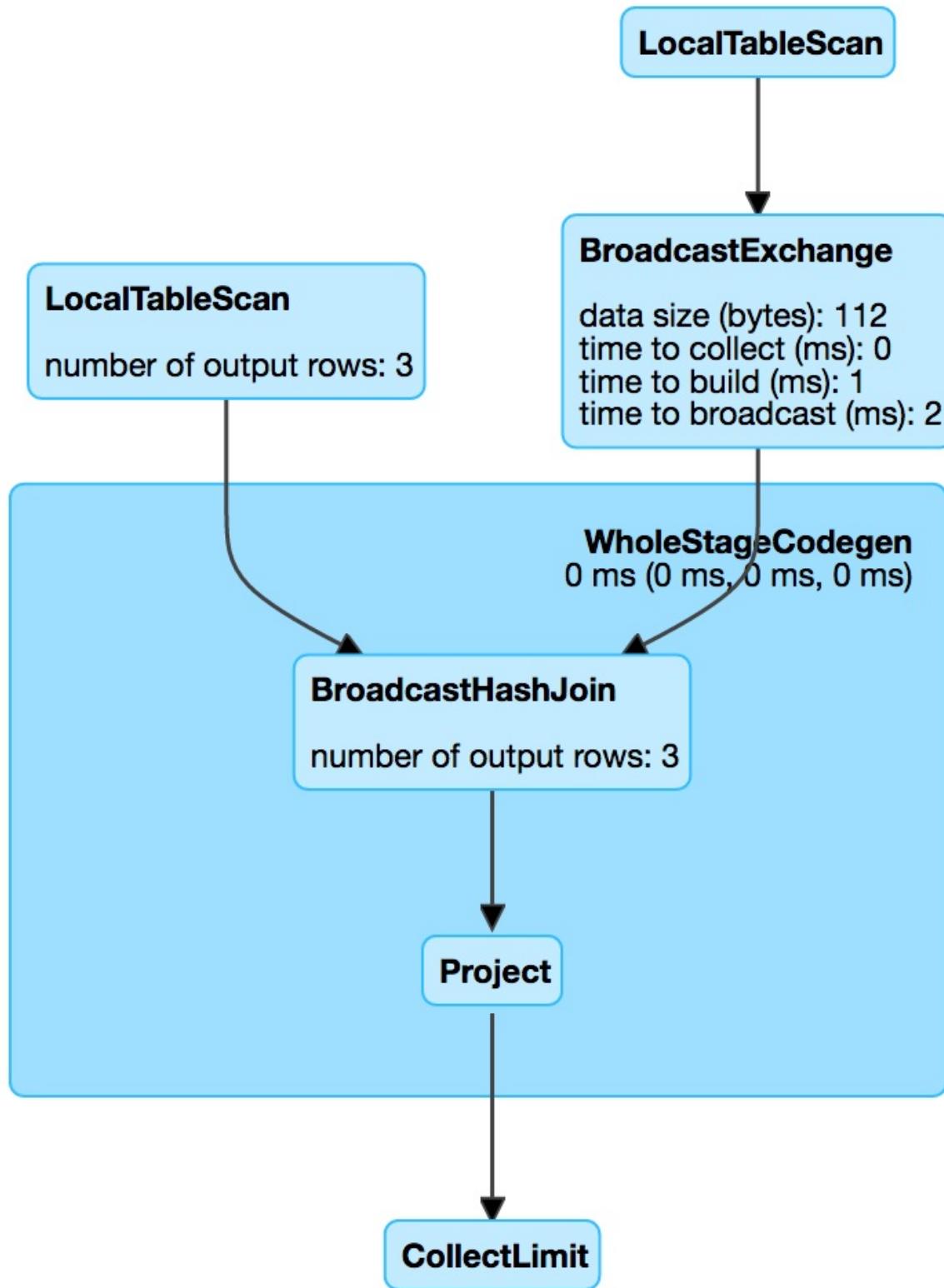


Figure 1. BroadcastHashJoinExec in web UI (Details for Query)

Note

The prefix for variable names for `BroadcastHashJoinExec` operators in [CodegenSupport](#)-generated code is **bhj**.

```

scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Project [id#15, token#16, token#21]
+- *BroadcastHashJoin [id#15], [id#20], Inner, BuildRight
  :- LocalTableScan [id#15, token#16]
  +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as
bigint)))
    +- LocalTableScan [id#20, token#21]

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */   return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {
/* 006 */   private Object[] references;
/* 007 */   private scala.collection.Iterator[] inputs;
/* 008 */   private scala.collection.Iterator inputadapter_input;
/* 009 */   private org.apache.spark.broadcast.TorrentBroadcast bhj_broadcast;
/* 010 */   private org.apache.spark.sql.execution.joins.LongHashedRelation bhj_relation;
/* 011 */   private org.apache.spark.sql.execution.metric.SQLMetric bhj_numOutputRows;
/* 012 */   private UnsafeRow bhj_result;
/* 013 */   private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder bhj_holder;
/* 014 */   private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter bhj_rowWriter;
...

```

Table 2. BroadcastHashJoinExec's Required Child Output Distributions

BuildSide	Left Child	Right Child
BuildLeft	BroadcastDistribution with HashedRelationBroadcastMode broadcast mode of build join keys	UnspecifiedDistribution
BuildRight	UnspecifiedDistribution	BroadcastDistribution with HashedRelationBroadcastMode broadcast mode of build join keys

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

```
doExecute ...FIXME
```

Generating Java Source Code for Inner Join

— `codegenInner` Internal Method

```
codegenInner(ctx: CodegenContext, input: Seq[ExprCode]): String
```

```
codegenInner ...FIXME
```

Note

`codegenInner` is used exclusively when `BroadcastHashJoinExec` is requested to generate the Java code for the "consume" path in whole-stage code generation.

Generating Java Source Code for Left or Right Outer Join

— `codegenOuter` Internal Method

```
codegenOuter(ctx: CodegenContext, input: Seq[ExprCode]): String
```

```
codegenOuter ...FIXME
```

Note

`codegenOuter` is used exclusively when `BroadcastHashJoinExec` is requested to generate the Java code for the "consume" path in whole-stage code generation.

Generating Java Source Code for Left Semi Join

— `codegenSemi` Internal Method

```
codegenSemi(ctx: CodegenContext, input: Seq[ExprCode]): String
```

```
codegenSemi ...FIXME
```

Note

`codegenSemi` is used exclusively when `BroadcastHashJoinExec` is requested to generate the Java code for the "consume" path in whole-stage code generation.

Generating Java Source Code for Anti Join

— `codegenAnti` Internal Method

```
codegenAnti(ctx: CodegenContext, input: Seq[ExprCode]): String
```

`codegenAnti` ...FIXME

Note

`codegenAnti` is used exclusively when `BroadcastHashJoinExec` is requested to generate the Java code for the "consume" path in whole-stage code generation.

`codegenExistence` Internal Method

```
codegenExistence(ctx: CodegenContext, input: Seq[ExprCode]): String
```

`codegenExistence` ...FIXME

Note

`codegenExistence` is used exclusively when `BroadcastHashJoinExec` is requested to generate the Java code for the "consume" path in whole-stage code generation.

`genStreamSideJoinKey` Internal Method

```
genStreamSideJoinKey(  
  ctx: CodegenContext,  
  input: Seq[ExprCode]): (ExprCode, String)
```

`genStreamSideJoinKey` ...FIXME

Note

`genStreamSideJoinKey` is used when `BroadcastHashJoinExec` is requested to generate the Java source code for `inner`, `outer`, `left semi`, `anti` and `existence` joins (for the "consume" path in whole-stage code generation).

Creating BroadcastHashJoinExec Instance

`BroadcastHashJoinExec` takes the following when created:

- Left join key [expressions](#)
- Right join key [expressions](#)
- [Join type](#)
- [BuildSide](#)

- Optional join condition [expression](#)
- Left [physical operator](#)
- Right [physical operator](#)

BroadcastNestedLoopJoinExec Binary Physical Operator

`BroadcastNestedLoopJoinExec` is a [binary physical operator](#) (with two child `left` and `right` physical operators) that is [created](#) (and converted to) when [JoinSelection](#) physical plan strategy finds a [Join](#) logical operator that meets either case:

- `canBuildRight` join type and `right` physical operator [broadcastable](#)
- `canBuildLeft` join type and `left broadcastable`
- non- `InnerLike` join type

Note	<code>BroadcastNestedLoopJoinExec</code> is the default physical operator when no other operators have matched selection requirements .
------	---

Note	<code>canBuildRight</code> join types are: <ul style="list-style-type: none"> • CROSS, INNER, LEFT ANTI, LEFT OUTER, LEFT SEMI or Existence <code>canBuildLeft</code> join types are: <ul style="list-style-type: none"> • CROSS, INNER, RIGHT OUTER
------	--

```

val nums = spark.range(2)
val letters = ('a' to 'c').map(_.toString).toDF("letter")
val q = nums.crossJoin(letters)

scala> q.explain
== Physical Plan ==
BroadcastNestedLoopJoin BuildRight, Cross
:- *Range (0, 2, step=1, splits=Some(8))
+- BroadcastExchange IdentityBroadcastMode
  +- LocalTableScan [letter#69]
  
```

Table 1. `BroadcastNestedLoopJoinExec`'s Performance Metrics

Key	Name (in web UI)	Description
<code>numOutputRows</code>	number of output rows	

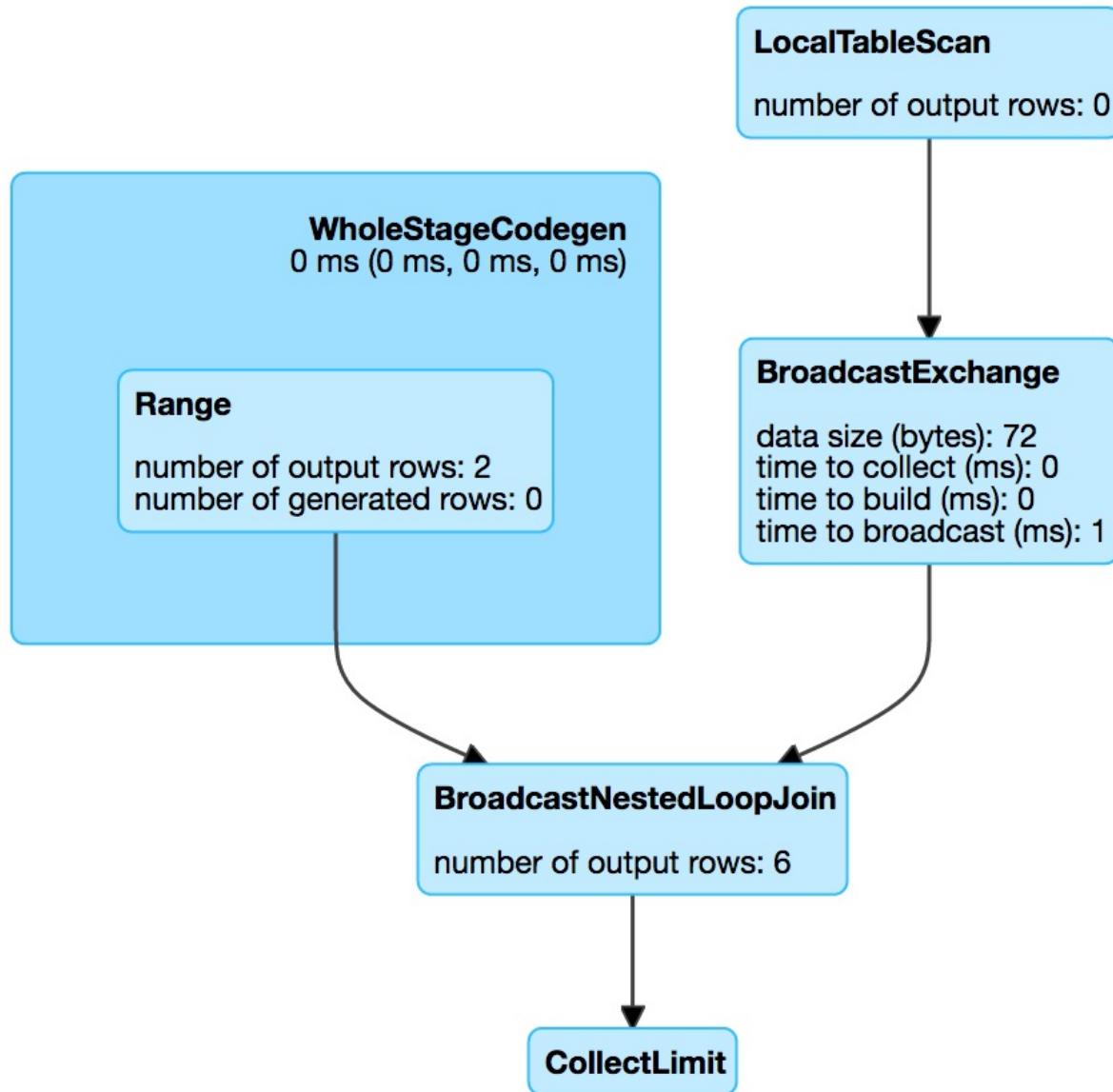


Figure 1. BroadcastNestedLoopJoinExec in web UI (Details for Query)

Table 2. BroadcastNestedLoopJoinExec's Required Child Output Distributions

BuildSide	Left Child	Right Child
BuildLeft	BroadcastDistribution (uses IdentityBroadcastMode broadcast mode)	UnspecifiedDistribution
BuildRight	UnspecifiedDistribution	BroadcastDistribution (uses IdentityBroadcastMode broadcast mode)

Creating BroadcastNestedLoopJoinExec Instance

BroadcastNestedLoopJoinExec takes the following when created:

- Left physical operator

- Right [physical operator](#)
- [BuildSide](#)
- [Join type](#)
- Optional join condition [expressions](#)

CartesianProductExec

CartesianProductExec is...FIXME

CoalesceExec Unary Physical Operator

`CoalesceExec` is a [unary physical operator](#) (i.e. with one `child` physical operator) to...

FIXME...with `numPartitions` number of partitions and a `child` spark plan.

`CoalesceExec` represents [Repartition](#) logical operator at execution (when shuffle was disabled — see [BasicOperators](#) execution planning strategy). When executed, it executes the input `child` and calls [coalesce](#) on the result RDD (with `shuffle` disabled).

Please note that since physical operators present themselves without the suffix *Exec*, `CoalesceExec` is the `Coalesce` in the Physical Plan section in the following example:

```
scala> df.rdd.getNumPartitions
res6: Int = 8

scala> df.coalesce(1).rdd.getNumPartitions
res7: Int = 1

scala> df.coalesce(1).explain(extended = true)
== Parsed Logical Plan ==
Repartition 1, false
+- LocalRelation [value#1]

== Analyzed Logical Plan ==
value: int
Repartition 1, false
+- LocalRelation [value#1]

== Optimized Logical Plan ==
Repartition 1, false
+- LocalRelation [value#1]

== Physical Plan ==
Coalesce 1
+- LocalTableScan [value#1]
```

`output` collection of [Attribute](#) matches the `child`'s (since `CoalesceExec` is about changing the number of partitions not the internal representation).

`outputPartitioning` returns a [SinglePartition](#) when the input `numPartitions` is `1` while a [UnknownPartitioning](#) partitioning scheme for the other cases.

DataSourceV2ScanExec Leaf Physical Operator

`DataSourceV2ScanExec` is a [leaf physical operator](#) that represents a [DataSourceV2Relation](#) logical operator at execution time.

Note	A DataSourceV2Relation logical operator is created exclusively when <code>DataFrameReader</code> is requested to "load" data (as a DataFrame) (from a data source with ReadSupport).
-------------	--

`DataSourceV2ScanExec` supports [ColumnarBatchScan](#) with [vectorized batch decoding](#) (when created for a [DataSourceReader](#) that supports it, i.e. the `DataSourceReader` is a [SupportsScanColumnarBatch](#) with the `enableBatchRead` flag enabled).

`DataSourceV2ScanExec` is also a [DataSourceV2StringFormat](#), i.e....FIXME

`DataSourceV2ScanExec` is [created](#) exclusively when [DataSourceV2Strategy](#) execution planning strategy is executed (i.e. applied to a logical plan) and finds a [DataSourceV2Relation](#) logical operator.

`DataSourceV2ScanExec` gives the single [input RDD](#) as the [only](#) input RDD of internal rows (when `WholeStageCodegenExec` physical operator is [executed](#)).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

<code>doExecute(): RDD[InternalRow]</code>	
--	--

Note	<code>doExecute</code> is part of SparkPlan Contract to generate the runtime representation of a structured query as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
-------------	---

`doExecute` ...FIXME

`supportsBatch` Property

<code>supportsBatch: Boolean</code>	
-------------------------------------	--

Note	<code>supportsBatch</code> is part of ColumnarBatchScan Contract to control whether the physical operator supports vectorized decoding or not.
-------------	--

`supportsBatch` is enabled (`true`) only when the [DataSourceReader](#) is a [SupportsScanColumnarBatch](#) with the [enableBatchRead](#) flag enabled.

Note	enableBatchRead flag is enabled by default.
------	---

`supportsBatch` is disabled (i.e. `false`) otherwise.

Creating DataSourceV2ScanExec Instance

`DataSourceV2ScanExec` takes the following when created:

- Output schema (as a collection of [AttributeReferences](#))
- [DataSourceReader](#)

`DataSourceV2ScanExec` initializes the [internal properties](#).

Creating Input RDD of Internal Rows — `inputRDD` Internal Property

`inputRDD: RDD[InternalRow]`

Note	<code>inputRDD</code> is a Scala lazy value which is computed once when accessed and cached afterwards.
------	---

`inputRDD` branches off per the type of the [DataSourceReader](#):

1. For a `continuousReader` in Spark Structured Streaming, `inputRDD` is a `ContinuousDataSourceRDD` that...FIXME
2. For a [SupportsScanColumnarBatch](#) with the [enableBatchRead](#) flag enabled, `inputRDD` is a [DataSourceRDD](#) with the [batchPartitions](#)
3. For all other types of the [DataSourceReader](#), `inputRDD` is a [DataSourceRDD](#) with the [partitions](#).

Note	<code>inputRDD</code> is used when <code>DataSourceV2ScanExec</code> physical operator is requested for the input RDDs and to execute .
------	---

Internal Properties

Name	Description
batchPartitions	Input partitions of ColumnarBatches (Seq[InputPartition[ColumnarBatch]])
partitions	Input partitions of InternalRows (Seq[InputPartition[InternalRow]])

DataWritingCommandExec Physical Operator

`DataWritingCommandExec` is a [physical operator](#) that is the execution environment for a `DataWritingCommand` logical command at [execution time](#).

`DataWritingCommandExec` is [created](#) exclusively when [BasicOperators](#) execution planning strategy is requested to plan a `DataWritingCommand` logical command.

When requested for [performance metrics](#), `DataWritingCommandExec` simply requests the `DataWritingCommand` for them.

Table 1. DataWritingCommandExec's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>sideEffectResult</code>	<p>Collection of InternalRows (<code>Seq[InternalRow]</code>) that is the result of executing the <code>DataWritingCommand</code> (with the SparkPlan)</p> <p>Used when <code>DataWritingCommandExec</code> is requested to executeCollect, executeToIterator, executeTake and doExecute</p>

Creating DataWritingCommandExec Instance

`DataWritingCommandExec` takes the following when created:

- `DataWritingCommand`
- Child [physical plan](#)

Executing Physical Operator and Collecting Results

— `executeCollect` Method

```
executeCollect(): Array[InternalRow]
```

Note

`executeCollect` is part of the [SparkPlan Contract](#) to execute the physical operator and collect results.

`executeCollect ...FIXME`

executeToIterator Method

```
executeToIterator: Iterator[InternalRow]
```

Note

`executeToIterator` is part of the [SparkPlan Contract](#) to...FIXME.

`executeToIterator` ...FIXME

Taking First N UnsafeRows — `executeTake` Method

```
executeTake(limit: Int): Array[InternalRow]
```

Note

`executeTake` is part of the [SparkPlan Contract](#) to take the first n `UnsafeRows`.

`executeTake` ...FIXME

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of the [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` simply requests the [SQLContext](#) for the [SparkContext](#) that is then requested to distribute (`parallelize`) the `sideEffectResult` (over 1 partition).

DebugExec Unary Physical Operator

DebugExec is a [unary physical operator](#) that...FIXME

dumpStats Method

```
dumpStats(): Unit
```

dumpStats ...FIXME

Note

dumpStats is used when...FIXME

DeserializeToObjectExec

DeserializeToObjectExec is...FIXME

ExecutedCommandExec Leaf Physical Operator for Command Execution

`ExecutedCommandExec` is a [leaf physical operator](#) for executing logical commands with side effects.

`ExecutedCommandExec` runs a command and caches the result in `sideEffectResult` internal attribute.

Table 1. ExecutedCommandExec's Methods

Method	Description
<code>doExecute</code>	Executes <code>ExecutedCommandExec</code> physical operator (and produces a result as an RDD of internal binary rows)
<code>executeCollect</code>	
<code>executeTake</code>	
<code>executeToIterator</code>	

Executing Logical RunnableCommand and Caching Result As InternalRows — `sideEffectResult` Internal Lazy Attribute

```
sideEffectResult: Seq[InternalRow]
```

`sideEffectResult` requests `RunnableCommand` to `run` (that produces a `Seq[Row]`) and converts the result to Catalyst types using a Catalyst converter function for the schema.

Note

`sideEffectResult` is used when `ExecutedCommandExec` is requested for `executeCollect`, `executeToIterator`, `executeTake`, `doExecute`.

ExpandExec

ExpandExec is...FIXME

ExternalRDDScanExec Leaf Physical Operator

ExternalRDDScanExec is a [leaf physical operator](#) for...FIXME

FileSourceScanExec Leaf Physical Operator

`FileSourceScanExec` is a [leaf physical operator](#) (as a [DataSourceScanExec](#)) that represents a scan over collections of files (incl. Hive tables).

`FileSourceScanExec` is [created](#) exclusively for a [LogicalRelation](#) logical operator with a [HadoopFsRelation](#) when [FileSourceStrategy](#) execution planning strategy is executed.

```
// Create a bucketed data source table
// It is one of the most complex examples of a LogicalRelation with a HadoopFsRelation
val tableName = "bucketed_4_id"
spark
  .range(100)
  .withColumn("part", $"id" % 2)
  .write
  .partitionBy("part")
  .bucketBy(4, "id")
  .sortBy("id")
  .mode("overwrite")
  .saveAsTable(tableName)
val q = spark.table(tableName)

val sparkPlan = q.queryExecution.executedPlan
scala> :type sparkPlan
org.apache.spark.sql.execution.SparkPlan

scala> println(sparkPlan.numberedTreeString)
00 *(1) FileScan parquet default.bucketed_4_id[id#7L,part#8L] Batched: true, Format: Parquet, Location: CatalogFileIndex[file:/Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id], PartitionCount: 2, PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:bigint>, SelectedBucketsCount: 4 out of 4

import org.apache.spark.sql.execution.FileSourceScanExec
val scan = sparkPlan.collectFirst { case exec: FileSourceScanExec => exec }.get

scala> :type scan
org.apache.spark.sql.execution.FileSourceScanExec

scala> scan.metadata.toSeq.sortBy(_.value).map { case (k, v) => s"$k -> $v" }.foreach(println)
Batched -> true
Format -> Parquet
Location -> CatalogFileIndex[file:/Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id]
PartitionCount -> 2
PartitionFilters -> []
PushedFilters -> []
ReadSchema -> struct<id:bigint>
SelectedBucketsCount -> 4 out of 4
```

`FileSourceScanExec` uses the single [input RDD](#) as the [input RDDs](#) (in [Whole-Stage Java Code Generation](#)).

When `executed`, `FileSourceScanExec` operator creates a [FileScanRDD](#) (for [bucketed](#) and [non-bucketed reads](#)).

```

scala> :type scan
org.apache.spark.sql.execution.FileSourceScanExec

val rdd = scan.execute
scala> println(rdd.toDebugString)
(6) MapPartitionsRDD[7] at execute at <console>:28 []
|  FileScanRDD[2] at execute at <console>:27 []

import org.apache.spark.sql.execution.datasources.FileScanRDD
assert(rdd.dependencies.head.rdd.isInstanceOf[FileScanRDD])

```

`FileSourceScanExec` supports [bucket pruning](#) so it only scans the bucket files required for a query.

```

scala> :type scan
org.apache.spark.sql.execution.FileSourceScanExec

import org.apache.spark.sql.execution.datasources.FileScanRDD
val rdd = scan.inputRDDs.head.asInstanceOf[FileScanRDD]

import org.apache.spark.sql.execution.datasources.FilePartition
val bucketFiles = for {
  FilePartition(bucketId, files) <- rdd.filePartitions
  f <- files
} yield s"Bucket $bucketId => $f"

scala> println(bucketFiles.size)
51

scala> bucketFiles.foreach(println)
Bucket 0 => path: file:///Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id/part=0/part-00004-5301d371-01c3-47d4-bb6b-76c3c94f3699_00000.c000.snappy.parquet, range: 0-423, partition values: [0]
Bucket 0 => path: file:///Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id/part=0/part-00001-5301d371-01c3-47d4-bb6b-76c3c94f3699_00000.c000.snappy.parquet, range: 0-423, partition values: [0]
...
Bucket 3 => path: file:///Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id/part=1/part-00005-5301d371-01c3-47d4-bb6b-76c3c94f3699_00003.c000.snappy.parquet, range: 0-423, partition values: [1]
Bucket 3 => path: file:///Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id/part=1/part-00000-5301d371-01c3-47d4-bb6b-76c3c94f3699_00003.c000.snappy.parquet, range: 0-431, partition values: [1]
Bucket 3 => path: file:///Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id/part=1/part-00007-5301d371-01c3-47d4-bb6b-76c3c94f3699_00003.c000.snappy.parquet, range: 0-423, partition values: [1]

```

`FileSourceScanExec` uses a [HashPartitioning](#) or the default [UnknownPartitioning](#) as the output partitioning scheme.

`FileSourceScanExec` is a `ColumnarBatchScan` and supports batch decoding only when the `FileFormat` (of the `HadoopFsRelation`) supports it.

`FileSourceScanExec` supports data source filters that are printed out to the console (at INFO logging level) and available as metadata (e.g. in web UI or `explain`).

```
Pushed Filters: [pushedDownFilters]
```

Table 1. FileSourceScanExec's Performance Metrics

Key	Name (in web UI)	Description
<code>metadataTime</code>	metadata time (ms)	
<code>numFiles</code>	number of files	
<code>numOutputRows</code>	number of output rows	
<code>scanTime</code>	scan time	

As a `DataSourceScanExec`, `FileSourceScanExec` uses **Scan** for the prefix of the node name.

```
val fileScanExec: FileSourceScanExec = ... // see the example earlier
assert(fileScanExec.nodeName startsWith "Scan")
```

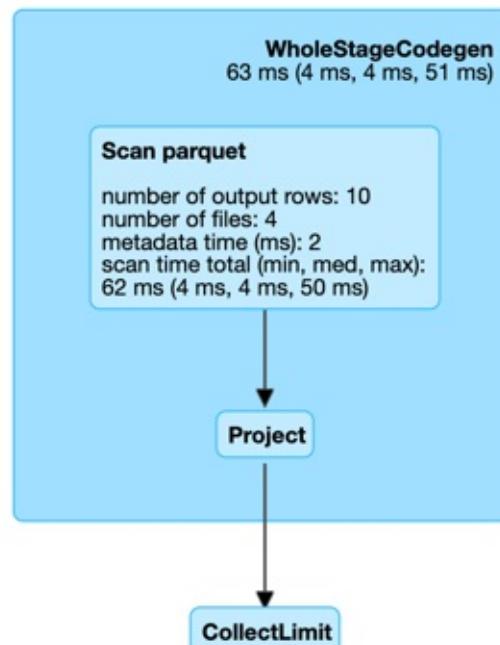


Figure 1. FileSourceScanExec in web UI (Details for Query)

`FileSourceScanExec` uses **File** for `nodeNamePrefix` (that is used for the `simple node description` in query plans).

```

val fileScanExec: FileSourceScanExec = ... // see the example earlier
assert(fileScanExec.nodeNamePrefix == "File")

scala> println(fileScanExec.simpleString)
FileScan csv [id#20,name#21,city#22] Batched: false, Format: CSV, Location: InMemoryFi
leIndex[file:/Users/jacek/dev/oss/datasets/people.csv], PartitionFilters: [], PushedFi
lters: [], ReadSchema: struct<id:string,name:string,city:string>

```

Table 2. FileSourceScanExec's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description	
metadata	<p>metadata: Map[String, String]</p> <p>Metadata</p> <p>Note metadata is part of DataSourceScanExec Contract to..FIXME.</p>	
pushedDownFilters	<p>Data source filters that are dataFilters expressions converted to their respective filters</p> <p>Tip Enable INFO logging level to see pushedDownFilters printed out to the console.</p> <p>Pushed Filters: [pushedDownFilters]</p> <p>Used when FileSourceScanExec is requested for the metadata and input RDD</p>	
Tip	<p>Enable ALL logging level for org.apache.spark.sql.execution.FileSourceScanExec logger to see what happens inside.</p> <p>Add the following line to conf/log4j.properties :</p> <pre>log4j.logger.org.apache.spark.sql.execution.FileSourceScanExec=ALL</pre> <p>Refer to Logging.</p>	

Creating RDD for Non-Bucketed Reads

— `createNonBucketedReadRDD` Internal Method

```
createNonBucketedReadRDD(
    readFile: (PartitionedFile) => Iterator[InternalRow],
    selectedPartitions: Seq[PartitionDirectory],
    fsRelation: HadoopFsRelation): RDD[InternalRow]
```

`createNonBucketedReadRDD` calculates the maximum size of partitions (`maxSplitBytes`) based on the following properties:

- `spark.sql.files.maxPartitionBytes` (default: `128m`)
- `spark.sql.files.openCostInBytes` (default: `4m`)

`createNonBucketedReadRDD` sums up the size of all the files (with the extra `spark.sql.files.openCostInBytes`) for the given `selectedPartitions` and divides the sum by the "default parallelism" (i.e. number of CPU cores assigned to a Spark application) that gives `bytesPerCore`.

The maximum size of partitions is then the minimum of `spark.sql.files.maxPartitionBytes` and the bigger of `spark.sql.files.openCostInBytes` and the `bytesPerCore`.

`createNonBucketedReadRDD` prints out the following INFO message to the logs:

```
Planning scan with bin packing, max size: [maxSplitBytes] bytes, open cost is considered as scanning [openCostInBytes] bytes.
```

For every file (as Hadoop's `FileStatus`) in every partition (as `PartitionDirectory` in the given `selectedPartitions`), `createNonBucketedReadRDD` gets the HDFS block locations to create `PartitionedFiles` (possibly split per the maximum size of partitions if the `FileFormat` of the `HadoopFsRelation` is `splittable`). The partitioned files are then sorted by number of bytes to read (aka *split size*) in decreasing order (from the largest to the smallest).

`createNonBucketedReadRDD` "compresses" multiple splits per partition if together they are smaller than the `maxSplitBytes` ("Next Fit Decreasing") that gives the necessary partitions (file blocks as `FilePartitions`).

In the end, `createNonBucketedReadRDD` creates a `FileScanRDD` (with the given `(PartitionedFile) → Iterator[InternalRow]` read function and the partitions).

Note

`createNonBucketedReadRDD` is used exclusively when `FileSourceScanExec` physical operator is requested for the `input RDD` (and neither the optional `bucketing specification` of the `HadoopFsRelation` is defined nor `bucketing is enabled`).

selectedPartitions Internal Lazy-Initialized Property

```
selectedPartitions: Seq[PartitionDirectory]
```

```
selectedPartitions ...FIXME
```

Note

- `selectedPartitions` is used when `FileSourceScanExec` is requested for the following:
 - `outputPartitioning` and `outputOrdering` when `bucketing is enabled` and the optional `bucketing specification` of the `HadoopFsRelation` is defined
 - `metadata`
 - `inputRDD`

Creating FileSourceScanExec Instance

`FileSourceScanExec` takes the following when created:

- `HadoopFsRelation`
- Output schema attributes
- Schema
- `partitionFilters` `expressions`
- Bucket IDs for bucket pruning (`Option[BitSet]`)
- `dataFilters` `expressions`
- Optional `TableIdentifier`

`FileSourceScanExec` initializes the `internal registries and counters`.

Output Partitioning Scheme — `outputPartitioning` Attribute

```
outputPartitioning: Partitioning
```

Note

`outputPartitioning` is part of the `SparkPlan Contract` to specify output data partitioning.

`outputPartitioning` can be one of the following:

- HashPartitioning (with the bucket column names and the number of buckets of the bucketing specification of the HadoopFsRelation) when bucketing is enabled and the HadoopFsRelation has a bucketing specification defined
- UnknownPartitioning (with 0 partitions) otherwise

Creating FileScanRDD with Bucketing Support

— `createBucketedReadRDD` Internal Method

```
createBucketedReadRDD(
    bucketSpec: BucketSpec,
    readFile: (PartitionedFile) => Iterator[InternalRow],
    selectedPartitions: Seq[PartitionDirectory],
    fsRelation: HadoopFsRelation): RDD[InternalRow]
```

`createBucketedReadRDD` prints the following INFO message to the logs:

```
Planning with [numBuckets] buckets
```

`createBucketedReadRDD` maps the available files of the input `selectedPartitions` into `PartitionedFiles`. For every file, `createBucketedReadRDD` `getBlockLocations` and `getBlockHosts`.

`createBucketedReadRDD` then groups the `PartitionedFiles` by bucket ID.

Note	Bucket ID is of the format <code>_0000n</code> , i.e. the bucket ID prefixed with up to four 0's.
------	---

`createBucketedReadRDD` prunes (filters out) the bucket files for the bucket IDs that are not listed in the [bucket IDs for bucket pruning](#).

`createBucketedReadRDD` creates a [FilePartition](#) (*file block*) for every bucket ID and the (pruned) bucket `PartitionedFiles`.

In the end, `createBucketedReadRDD` creates a [FileScanRDD](#) (with the input `readFile` for the `read` function and the file blocks (`FilePartitions`) for every bucket ID for `partitions`)

Use `RDD.toDebugString` to see `FileScanRDD` in the RDD execution plan (aka RDD lineage).

Tip

```
// Create a bucketed table
spark.range(8).write.bucketBy(4, "id").saveAsTable("b1")

scala> sql("desc extended b1").where($"col_name" like "%Bucket%").show
+-----+-----+
|      col_name|data_type|comment|
+-----+-----+
|    Num Buckets|        4|      |
|Bucket Columns| [`id`]|      |
+-----+-----+

val bucketedTable = spark.table("b1")

val lineage = bucketedTable.queryExecution.toRdd.toDebugString
scala> println(lineage)
(4) MapPartitionsRDD[26] at toRdd at <console>:26 []
 |  FileScanRDD[25] at toRdd at <console>:26 []
```

Note

`createBucketedReadRDD` is used exclusively when `FileSourceScanExec` physical operator is requested for the `inputRDD` (and the optional `bucketing` specification of the `HadoopFsRelation` is defined and `bucketing` is enabled).

supportsBatch Attribute

`supportsBatch: Boolean`

Note

`supportsBatch` is part of the [ColumnarBatchScan Contract](#) to enable [vectorized decoding](#).

`supportsBatch` is enabled (`true`) only when the `FileFormat` (of the `HadoopFsRelation`) supports [vectorized decoding](#). Otherwise, `supportsBatch` is disabled (i.e. `false`).

Note

`FileFormat` does not support vectorized decoding by default (i.e. `supportBatch` flag is disabled). Only `ParquetFileFormat` and `OrcFileFormat` have support for it under certain conditions.

FileSourceScanExec As ColumnarBatchScan

`FileSourceScanExec` is a [ColumnarBatchScan](#) and [supports batch decoding](#) only when the `FileFormat` (of the `HadoopFsRelation`) [supports it](#).

`FileSourceScanExec` has `needsUnsafeRowConversion` flag enabled for `ParquetFileFormat` data sources exclusively.

`FileSourceScanExec` has `vectorTypes`...FIXME

needsUnsafeRowConversion Flag

`needsUnsafeRowConversion: Boolean`

Note

`needsUnsafeRowConversion` is part of [ColumnarBatchScan Contract](#) to control the name of the variable for an input row while generating the Java source code to consume generated columns or row from a physical operator.

`needsUnsafeRowConversion` is enabled (i.e. `true`) when the following conditions all hold:

1. `FileFormat` of the [HadoopFsRelation](#) is `ParquetFileFormat`
2. `spark.sql.parquet.enableVectorizedReader` configuration property is enabled (default: `true`)

Otherwise, `needsUnsafeRowConversion` is disabled (i.e. `false`).

Note

`needsUnsafeRowConversion` is used when `FileSourceScanExec` is [executed](#) (and `supportsBatch` flag is off).

Fully-Qualified Class Names (Types) of Concrete ColumnVectors — vectorTypes Method

`vectorTypes: Option[Seq[String]]`

Note

`vectorTypes` is part of [ColumnarBatchScan Contract](#) to..FIXME.

`vectorTypes` simply requests the `FileFormat` of the [HadoopFsRelation](#) for `vectorTypes`.

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

`doExecute(): RDD[InternalRow]`

Note

`doExecute` is part of the [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` branches off per `supportsBatch` flag.

Note

`supportsBatch` flag can be enabled for `ParquetFileFormat` and `OrcFileFormat` built-in file formats (under certain conditions).

With `supportsBatch` flag enabled, `doExecute` creates a `WholeStageCodegenExec` physical operator (with the `FileSourceScanExec` for the child physical operator and `codegenStageId` as `0`) and executes it right after.

With `supportsBatch` flag disabled, `doExecute` creates an `unsafeRows` RDD to scan over which is different per `needsUnsafeRowConversion` flag.

If `needsUnsafeRowConversion` flag is on, `doExecute` takes the `inputRDD` and creates a new RDD by applying a function to each partition (using `RDD.mapPartitionsWithIndexInternal`):

1. Creates a `UnsafeProjection` for the `schema`
2. Initializes the `UnsafeProjection`
3. Maps over the rows in a partition iterator using the `UnsafeProjection` projection

Otherwise, `doExecute` simply takes the `inputRDD` as the `unsafeRows` RDD (with no changes).

`doExecute` takes the `numOutputRows` metric and creates a new RDD by mapping every element in the `unsafeRows` and incrementing the `numOutputRows` metric.

Tip

Use `RDD.toDebugString` to review the RDD lineage and "reverse-engineer" the values of the `supportsBatch` and `needsUnsafeRowConversion` flags given the number of RDDs.

With `supportsBatch` off and `needsUnsafeRowConversion` on you should see two more RDDs in the RDD lineage.

Creating Input RDD of Internal Rows — `inputRDD` Internal Property

`inputRDD: RDD[InternalRow]`

Note

`inputRDD` is a Scala lazy value which is computed once when accessed and cached afterwards.

`inputRDD` is an input `RDD` of `internal binary rows` (i.e. `InternalRow`) that is used when `FileSourceScanExec` physical operator is requested for `inputRDDs` and `execution`.

When created, `inputRDD` requests `HadoopFsRelation` to get the underlying `FileFormat` that is in turn requested to [build a data reader with partition column values appended](#) (with the input parameters from the properties of `HadoopFsRelation` and `pushedDownFilters`).

In case `HadoopFsRelation` has [bucketing specification defined](#) and [bucketing support is enabled](#), `inputRDD` creates a `FileScanRDD` with [bucketing](#) (with the bucketing specification, the reader, `selectedPartitions` and the `HadoopFsRelation` itself). Otherwise, `inputRDD` creates `createNonBucketedReadRDD`.

Note

`createBucketedReadRDD` accepts a bucketing specification while `createNonBucketedReadRDD` does not.

Output Data Ordering — `outputOrdering` Attribute

```
outputOrdering: Seq[SortOrder]
```

Note

`outputOrdering` is part of the [SparkPlan Contract](#) to specify output data ordering.

`outputOrdering` is a `SortOrder` expression for every `sort column` in `Ascending` order only when all the following hold:

- [bucketing is enabled](#)
- `HadoopFsRelation` has a [bucketing specification defined](#)
- All the buckets have a single file in it

Otherwise, `outputOrdering` is simply empty (`Nil`).

updateDriverMetrics Internal Method

```
updateDriverMetrics(): Unit
```

`updateDriverMetrics` updates the following [performance metrics](#):

- `numFiles` metric with the total of all the sizes of the files in the `selectedPartitions`
- `metadataTime` metric with the time spent in the `selectedPartitions`

In the end, `updateDriverMetrics` requests the `SQLMetrics` object to [posts the metric updates](#).

Note

`updateDriverMetrics` is used exclusively when `FileSourceScanExec` physical operator is requested for the `input RDD` (the very first time).

getBlockLocations Internal Method

```
getBlockLocations(file: FileStatus): Array[BlockLocation]
```

`getBlockLocations` simply requests the given Hadoop `FileStatus` for the block locations (`getBlockLocations`) if it is a Hadoop `LocatedFileStatus`. Otherwise, `getBlockLocations` returns an empty array.

Note

`getBlockLocations` is used when `FileSourceScanExec` physical operator is requested to `createBucketedReadRDD` and `createNonBucketedReadRDD`.

FilterExec Unary Physical Operator

`FilterExec` is a [unary physical operator](#) (i.e. with one `child` physical operator) that represents `Filter` and `TypedFilter` unary logical operators at execution.

`FilterExec` supports [Java code generation](#) (aka `codegen`) as follows:

- `usedInputs` is an empty `Attributeset` (to defer evaluation of attribute expressions until they are actually used, i.e. in the [generated Java source code for consume path](#))
- Uses whatever the `child` physical operator uses for the `input RDDs`
- Generates a Java source code for the `produce` and `consume` paths in whole-stage code generation

`FilterExec` is [created](#) when:

- `BasicOperators` execution planning strategy is [executed](#) (and plans `Filter` and `TypedFilter` unary logical operators)
- `HiveTableScans` execution planning strategy is [executed](#) (and plans `HiveTableRelation` leaf logical operators and requests `SparkPlanner` to [pruneFilterProject](#))
- `InMemoryScans` execution planning strategy is [executed](#) (and plans `InMemoryRelation` leaf logical operators and requests `SparkPlanner` to [pruneFilterProject](#))
- `DataSourceStrategy` execution planning strategy is requested to [create](#) a `RowDataSourceScanExec` physical operator (possibly under `FilterExec` and `ProjectExec` operators)
- `FileSourceStrategy` execution planning strategy is [executed](#) (on `LogicalRelations` with a `HadoopFsRelation`)
- `ExtractPythonUDFs` physical query optimization is requested to [trySplitFilter](#)

Table 1. FilterExec's Performance Metrics

Key	Name (in web UI)	Description
<code>numOutputRows</code>	number of output rows	

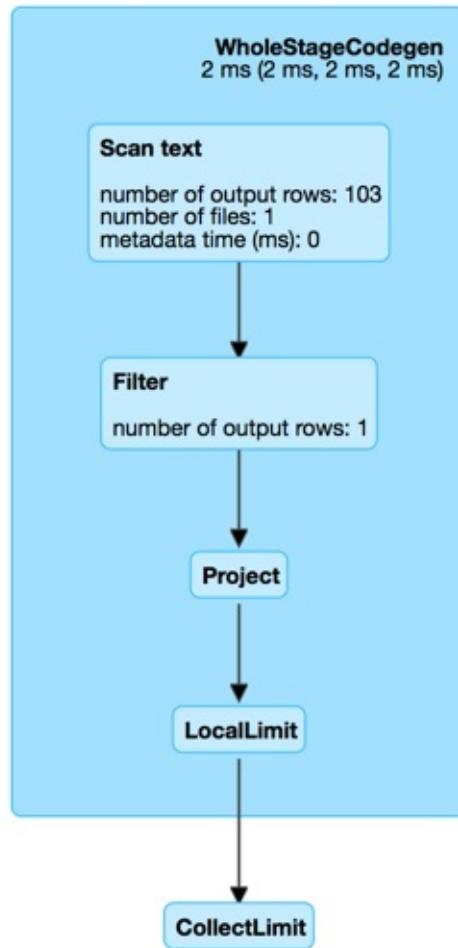


Figure 1. FilterExec in web UI (Details for Query)

`FilterExec` uses whatever the `child` physical operator uses for the `input RDDs`, the `outputOrdering` and the `outputPartitioning`.

`FilterExec` uses the `PredicateHelper` for...FIXME

Table 2. FilterExec's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
notNullAttributes	FIXME Used when...FIXME
notNullPreds	FIXME Used when...FIXME
otherPreds	FIXME Used when...FIXME

Creating FilterExec Instance

`FilterExec` takes the following when created:

- Catalyst expression for the filter condition
- Child physical operator

`FilterExec` initializes the [internal registries and counters](#).

isNullIntolerant Internal Method

```
isNullIntolerant(expr: Expression): Boolean
```

`isNullIntolerant` ...FIXME

Note	<code>isNullIntolerant</code> is used when...FIXME
------	--

usedInputs Method

```
usedInputs: AttributeSet
```

Note	<code>usedInputs</code> is part of CodegenSupport Contract to...FIXME.
------	--

`usedInputs` ...FIXME

output Method

```
output: Seq[Attribute]
```

Note	<code>output</code> is part of QueryPlan Contract to...FIXME.
------	---

`output` ...FIXME

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — doProduce Method

```
doProduce(ctx: CodegenContext): String
```

Note	<code>doProduce</code> is part of CodegenSupport Contract to generate the Java source code for <code>produce path</code> in Whole-Stage Code Generation.
------	--

```
doProduce ...FIXME
```

Generating Java Source Code for Consume Path in Whole-Stage Code Generation — `doConsume` Method

```
doConsume(ctx: CodegenContext, input: Seq[ExprCode], row: ExprCode): String
```

Note

`doConsume` is part of [CodegenSupport Contract](#) to generate the Java source code for [consume path](#) in Whole-Stage Code Generation.

`doConsume` creates a new [metric term](#) for the [numOutputRows](#) metric.

```
doConsume ...FIXME
```

In the end, `doConsume` uses [consume](#) and [FIXME](#) to generate a Java source code (as a plain text) inside a `do {...} while(false);` code block.

```
// DEMO Write one
```

genPredicate Internal Method

```
genPredicate(c: Expression, in: Seq[ExprCode], attrs: Seq[Attribute]): String
```

Note

`genPredicate` is an internal method of [doConsume](#).

```
genPredicate ...FIXME
```

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` executes the [child](#) physical operator and creates a new [MapPartitionsRDD](#) that does the filtering.

```
// DEMO Show the RDD lineage with the new MapPartitionsRDD after FilterExec
```

Internally, `doExecute` takes the `numOutputRows` metric.

In the end, `doExecute` requests the `child` physical operator to `execute` (that triggers physical query planning and generates an `RDD[InternalRow]`) and transforms it by executing the following function on internal rows per partition with index (using `RDD.mapPartitionsWithIndexInternal` that creates another RDD):

1. Creates a partition filter as a new `GenPredicate` (for the `filter condition` expression and the `output schema` of the `child` physical operator)
2. Requests the generated partition filter `Predicate` to `initialize` (with `0` partition index)
3. Filters out elements from the partition iterator (`Iterator[InternalRow]`) by requesting the generated partition filter `Predicate` to evaluate for every `InternalRow`
 - i. Increments the `numOutputRows` metric for positive evaluations (i.e. that returned `true`)

Note

`doExecute` (by `RDD.mapPartitionsWithIndexInternal`) adds a new `MapPartitionsRDD` to the RDD lineage. Use `RDD.toDebugString` to see the additional `MapPartitionsRDD`.

GenerateExec Unary Physical Operator

`GenerateExec` is a [unary physical operator](#) (i.e. with one `child` physical operator) that is [created](#) exclusively when `BasicOperators` execution planning strategy is requested to resolve a [Generate logical operator](#).

```

val nums = Seq((0 to 4).toArray).toDF("nums")
val q = nums.withColumn("explode", explode($"nums"))

scala> q.explain
== Physical Plan ==
Generate explode(nums#3), true, false, [explode#12]
+- LocalTableScan [nums#3]

val sparkPlan = q.queryExecution.executedPlan
import org.apache.spark.sql.execution.GenerateExec
val ge = sparkPlan.asInstanceOf[GenerateExec]

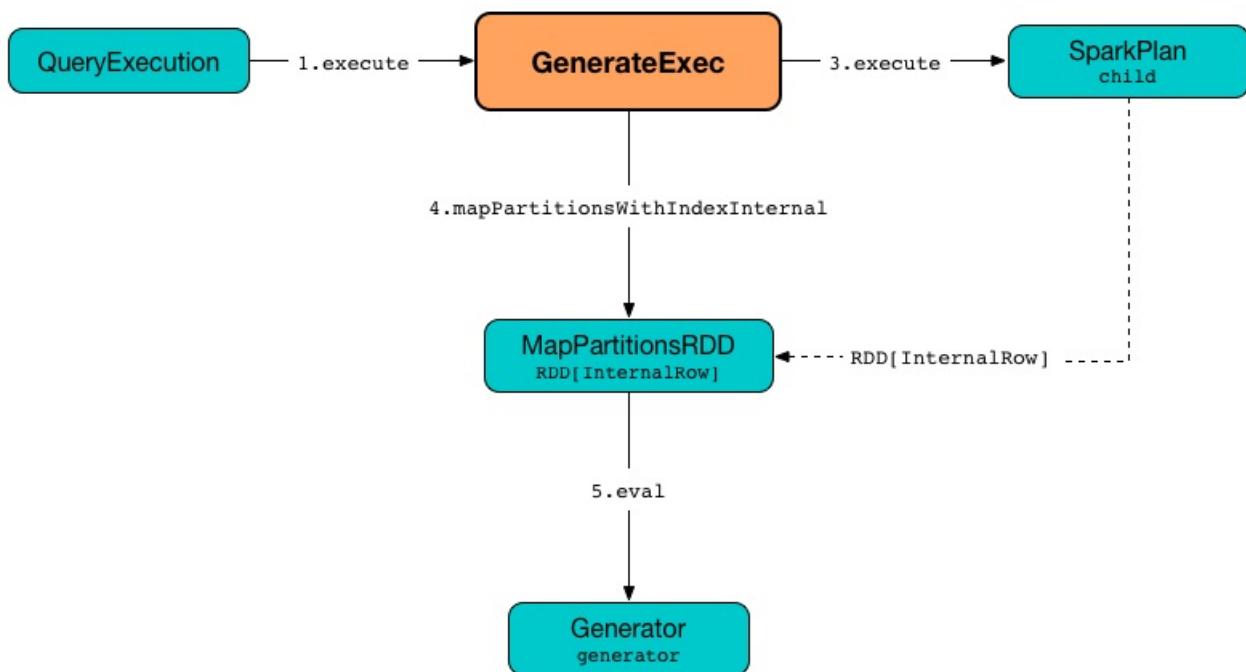
scala> :type ge
org.apache.spark.sql.execution.GenerateExec

val rdd = ge.execute

scala> rdd.toDebugString
res1: String =
(1) MapPartitionsRDD[2] at execute at <console>:26 []
 | MapPartitionsRDD[1] at execute at <console>:26 []
 | ParallelCollectionRDD[0] at execute at <console>:26 []

```

When [executed](#), `GenerateExec` [executes](#) (aka [evaluates](#)) the `Generator` expression on every row in a RDD partition.

Figure 1. GenerateExec's Execution — `doExecute` Method

Note

`child` physical operator has to support [CodegenSupport](#).

`GenerateExec` supports [Java code generation](#) (aka *codegen*).

`GenerateExec` does not support [Java code generation](#) (aka *whole-stage codegen*), i.e. `supportCodegen` flag is turned off.

```

scala> :type ge
org.apache.spark.sql.execution.GenerateExec

scala> ge.supportCodegen
res2: Boolean = false
  
```

```

// Turn spark.sql.codegen.comments on to see comments in the code
// ./bin/spark-shell --conf spark.sql.codegen.comments=true
// inline function gives Inline expression
val q = spark.range(1)
  .selectExpr("inline(array(struct(1, 'a'), struct(2, 'b')))")

scala> q.explain
== Physical Plan ==
Generate inline([[1,a],[2,b]]), false, false, [col1#47, col2#48]
+- *Project
  +- *Range (0, 1, step=1, splits=8)

val sparkPlan = q.queryExecution.executedPlan
import org.apache.spark.sql.execution.GenerateExec
val ge = sparkPlan.asInstanceOf[GenerateExec]

import org.apache.spark.sql.execution.WholeStageCodegenExec
  
```

```

val wsce = ge.child.asInstanceOf[WholeStageCodegenExec]
val (_, code) = wsce.doCodeGen
import org.apache.spark.sql.catalyst.expressions.codegen.CodeFormatter
val formattedCode = CodeFormatter.format(code)
scala> println(formattedCode)
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ /**
 * Codegend pipeline for
 * Project
 * +- Range (0, 1, step=1, splits=8)
 */
/* 006 */ final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {
/* 007 */     private Object[] references;
/* 008 */     private scala.collection.Iterator[] inputs;
/* 009 */     private org.apache.spark.sql.execution.metric.SQLMetric range_numOutputRows;
/* 010 */     private boolean range_initRange;
/* 011 */     private long range_number;
/* 012 */     private TaskContext range_taskContext;
/* 013 */     private InputMetrics range_inputMetrics;
/* 014 */     private long range_batchEnd;
/* 015 */     private long range_numElementsTodo;
/* 016 */     private scala.collection.Iterator range_input;
/* 017 */     private UnsafeRow range_result;
/* 018 */     private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder range_holder;
/* 019 */     private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter range_rowWriter;
/* 020 */
/* 021 */     public GeneratedIterator(Object[] references) {
/* 022 */         this.references = references;
/* 023 */     }
/* 024 */
/* 025 */     public void init(int index, scala.collection.Iterator[] inputs) {
/* 026 */         partitionIndex = index;
/* 027 */         this.inputs = inputs;
/* 028 */         range_numOutputRows = (org.apache.spark.sql.execution.metric.SQLMetric) references[0];
/* 029 */         range_initRange = false;
/* 030 */         range_number = 0L;
/* 031 */         range_taskContext = TaskContext.get();
/* 032 */         range_inputMetrics = range_taskContext.taskMetrics().inputMetrics();
/* 033 */         range_batchEnd = 0;
/* 034 */         range_numElementsTodo = 0L;
/* 035 */         range_input = inputs[0];
/* 036 */         range_result = new UnsafeRow(1);
/* 037 */         range_holder = new org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder(range_result, 0);
/* 038 */         range_rowWriter = new org.apache.spark.sql.catalyst.expressions.codegen.

```

```

UnsafeRowWriter(range_holder, 1);

/* 039 */
/* 040 */
/* 041 */
/* 042 */ private void initRange(int idx) {
/* 043 */     java.math.BigInteger index = java.math.BigInteger.valueOf(idx);
/* 044 */     java.math.BigInteger numSlice = java.math.BigInteger.valueOf(8L);
/* 045 */     java.math.BigInteger numElement = java.math.BigInteger.valueOf(1L);
/* 046 */     java.math.BigInteger step = java.math.BigInteger.valueOf(1L);
/* 047 */     java.math.BigInteger start = java.math.BigInteger.valueOf(0L);
/* 048 */     long partitionEnd;
/* 049 */
/* 050 */     java.math.BigInteger st = index.multiply(numElement).divide(numSlice).mu
ltiply(step).add(start);
/* 051 */     if (st.compareTo(java.math.BigInteger.valueOf(Long.MAX_VALUE)) > 0) {
/* 052 */         range_number = Long.MAX_VALUE;
/* 053 */     } else if (st.compareTo(java.math.BigInteger.valueOf(Long.MIN_VALUE)) < 0
) {
/* 054 */         range_number = Long.MIN_VALUE;
/* 055 */     } else {
/* 056 */         range_number = st.longValue();
/* 057 */     }
/* 058 */     range_batchEnd = range_number;
/* 059 */
/* 060 */     java.math.BigInteger end = index.add(java.math.BigInteger.ONE).multiply(
numElement).divide(numSlice)
/* 061 */         .multiply(step).add(start);
/* 062 */     if (end.compareTo(java.math.BigInteger.valueOf(Long.MAX_VALUE)) > 0) {
/* 063 */         partitionEnd = Long.MAX_VALUE;
/* 064 */     } else if (end.compareTo(java.math.BigInteger.valueOf(Long.MIN_VALUE)) <
0) {
/* 065 */         partitionEnd = Long.MIN_VALUE;
/* 066 */     } else {
/* 067 */         partitionEnd = end.longValue();
/* 068 */     }
/* 069 */
/* 070 */     java.math.BigInteger startToEnd = java.math.BigInteger.valueOf(partition
End).subtract(
/* 071 */         java.math.BigInteger.valueOf(range_number));
/* 072 */     range_numElementsTodo = startToEnd.divide(step).longValue();
/* 073 */     if (range_numElementsTodo < 0) {
/* 074 */         range_numElementsTodo = 0;
/* 075 */     } else if (startToEnd.remainder(step).compareTo(java.math.BigInteger.val
ueOf(0L)) != 0) {
/* 076 */         range_numElementsTodo++;
/* 077 */     }
/* 078 */ }
/* 079 */
/* 080 */ protected void processNext() throws java.io.IOException {
/* 081 */     // PRODUCE: Project
/* 082 */     // PRODUCE: Range (0, 1, step=1, splits=8)
/* 083 */     // initialize Range
/* 084 */     if (!range_initRange) {

```

```

/* 085 */     range_initRange = true;
/* 086 */     initRange(partitionIndex);
/* 087 */
/* 088 */
/* 089 */     while (true) {
/* 090 */         long range_range = range_batchEnd - range_number;
/* 091 */         if (range_range != 0L) {
/* 092 */             int range_localEnd = (int)(range_range / 1L);
/* 093 */             for (int range_localIdx = 0; range_localIdx < range_localEnd; range_
localIdx++) {
/* 094 */                 long range_value = ((long)range_localIdx * 1L) + range_number;
/* 095 */
/* 096 */                 // CONSUME: Project
/* 097 */                 // CONSUME: WholeStageCodegen
/* 098 */                 append(unsafeRow);
/* 099 */
/* 100 */                 if (shouldStop()) { range_number = range_value + 1L; return; }
/* 101 */             }
/* 102 */             range_number = range_batchEnd;
/* 103 */         }
/* 104 */
/* 105 */         range_taskContext.killTaskIfInterrupted();
/* 106 */
/* 107 */         long range_nextBatchTodo;
/* 108 */         if (range_numElementsTodo > 1000L) {
/* 109 */             range_nextBatchTodo = 1000L;
/* 110 */             range_numElementsTodo -= 1000L;
/* 111 */         } else {
/* 112 */             range_nextBatchTodo = range_numElementsTodo;
/* 113 */             range_numElementsTodo = 0;
/* 114 */             if (range_nextBatchTodo == 0) break;
/* 115 */         }
/* 116 */         range_numOutputRows.add(range_nextBatchTodo);
/* 117 */         range_inputMetrics.incRecordsRead(range_nextBatchTodo);
/* 118 */
/* 119 */         range_batchEnd += range_nextBatchTodo * 1L;
/* 120 */     }
/* 121 */ }
/* 122 */
/* 123 */ }
```

The [output schema](#) of a `GenerateExec` is...FIXME

Table 1. GenerateExec's Performance Metrics

Key	Name (in web UI)	Description
<code>numOutputRows</code>	number of output rows	

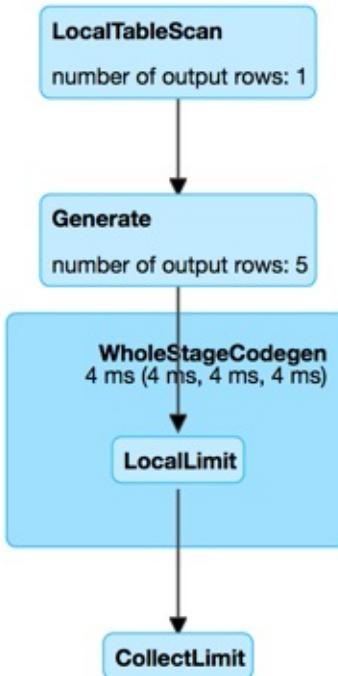


Figure 2. GenerateExec in web UI (Details for Query)

`producedAttributes ...FIXME`

`outputPartitioning ...FIXME`

`boundGenerator ...FIXME`

`GenerateExec` gives child's input RDDs (when `WholeStageCodegenExec` is executed).

`GenerateExec` requires that...FIXME

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

`doProduce(ctx: CodegenContext): String`

Note

`doProduce` is part of [CodegenSupport Contract](#) to generate the Java source code for [produce path](#) in Whole-Stage Code Generation.

`doProduce ...FIXME`

Generating Java Source Code for Consume Path in Whole-Stage Code Generation — `doConsume` Method

`doConsume(ctx: CodegenContext, input: Seq[ExprCode], row: ExprCode): String`

Note

`doConsume` is part of [CodegenSupport Contract](#) to generate the Java source code for [consume path](#) in Whole-Stage Code Generation.

`doConsume` ...FIXME

codeGenCollection Internal Method

```
codegenCollection(
  ctx: CodegenContext,
  e: CollectionGenerator,
  input: Seq[ExprCode],
  row: ExprCode): String
```

`codegenCollection` ...FIXME

Note

`codegenCollection` is used exclusively when `GenerateExec` is requested to [generate the Java code for the "consume" path in whole-stage code generation](#) (when `Generator` is a [CollectionGenerator](#)).

codeGenTraversableOnce Internal Method

```
codegenTraversableOnce(
  ctx: CodegenContext,
  e: Expression,
  input: Seq[ExprCode],
  row: ExprCode): String
```

`codegenTraversableOnce` ...FIXME

Note

`codegenTraversableOnce` is used exclusively when `GenerateExec` is requested to [generate the Java code for the consume path in whole-stage code generation](#) (when `Generator` is not a [CollectionGenerator](#)).

codeGenAccessor Internal Method

```
codegenAccessor(
  ctx: CodegenContext,
  source: String,
  name: String,
  index: String,
  dt: DataType,
  nullable: Boolean,
  initialChecks: Seq[String]): ExprCode
```

```
codegenAccessor ...FIXME
```

Note	codegenAccessor is used...FIXME
------	---------------------------------

Creating GenerateExec Instance

GenerateExec takes the following when created:

- Generator
- join flag
- outer flag
- Generator's output schema
- Child physical operator

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

```
doExecute(): RDD[InternalRow]
```

Note	doExecute is part of SparkPlan Contract to generate the runtime representation of a structured query as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	--

```
doExecute ...FIXME
```

HashAggregateExec Aggregate Physical Operator for Hash-Based Aggregation

`HashAggregateExec` is a [unary physical operator](#) (i.e. with one `child` physical operator) for **hash-based aggregation** that is [created](#) (indirectly through `AggUtils.createAggregate`) when:

- [Aggregation](#) execution planning strategy selects the aggregate physical operator for an [Aggregate](#) logical operator
- Structured Streaming's `StatefulAggregationStrategy` strategy creates plan for streaming `EventTimeWatermark` or [Aggregate](#) logical operators

Note

`HashAggregateExec` is the [preferred aggregate physical operator](#) for [Aggregation](#) execution planning strategy (over `ObjectHashAggregateExec` and `SortAggregateExec`).

`HashAggregateExec` supports [Java code generation](#) (aka `codegen`).

`HashAggregateExec` uses [TungstenAggregationIterator](#) (to iterate over `UnsafeRows` in partitions) when [executed](#).

```
val q = spark.range(10).
    groupBy('id % 2 as "group").
    agg(sum("id") as "sum")

// HashAggregateExec selected due to:
// 1. sum uses mutable types for aggregate expression
// 2. just a single id column reference of LongType data type
scala> q.explain
== Physical Plan ==
*HashAggregate(keys=[(id#0L % 2)#12L], functions=[sum(id#0L)])
+- Exchange hashpartitioning((id#0L % 2)#12L, 200)
   +- *HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#12L], functions=[partial_sum(id#0L)])
      +- *Range (0, 10, step=1, splits=8)

val execPlan = q.queryExecution.sparkPlan
scala> println(execPlan.numberedTreeString)
00 HashAggregate(keys=[(id#0L % 2)#15L], functions=[sum(id#0L)], output=[group#3L, sum#7L])
01 +- HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#15L], functions=[partial_sum(id#0L)], output=[(id#0L % 2)#15L, sum#17L])
02   +- Range (0, 10, step=1, splits=8)

// Going low level...watch your steps :)
```

```

import q.queryExecution.optimizedPlan
import org.apache.spark.sql.catalyst.plans.logical.Aggregate
val aggLog = optimizedPlan.asInstanceOf[Aggregate]
import org.apache.spark.sql.catalyst.planning.PhysicalAggregation
import org.apache.spark.sql.catalyst.expressions.aggregate.AggregateExpression
val aggregateExpressions: Seq[AggregateExpression] = PhysicalAggregation.unapply(aggLo
g).get._2
val aggregateBufferAttributes = aggregateExpressions.
  flatMap(_.aggregateFunction.aggBufferAttributes)
import org.apache.spark.sql.execution.aggregate.HashAggregateExec
// that's the exact reason why HashAggregateExec was selected
// Aggregation execution planning strategy prefers HashAggregateExec
scala> val useHash = HashAggregateExec.supportsAggregate(aggregateBufferAttributes)
useHash: Boolean = true

val hashAggExec = execPlan.asInstanceOf[HashAggregateExec]
scala> println(execPlan.numberedTreeString)
00 HashAggregate(keys=[(id#0L % 2)#15L], functions=[sum(id#0L)], output=[group#3L, sum#
7L])
01 +- HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#15L], functions=[partial_sum(id#0
L)], output=[(id#0L % 2)#15L, sum#17L])
02   +- Range (0, 10, step=1, splits=8)

val hashAggExecRDD = hashAggExec.execute // <-- calls doExecute
scala> println(hashAggExecRDD.toDebugString)
(8) MapPartitionsRDD[3] at execute at <console>:30 []
| MapPartitionsRDD[2] at execute at <console>:30 []
| MapPartitionsRDD[1] at execute at <console>:30 []
| ParallelCollectionRDD[0] at execute at <console>:30 []

```

Table 1. HashAggregateExec's Performance Metrics

Key	Name (in web UI)	Description
aggTime	aggregate time	
avgHashProbe	avg hash probe	<p>Average hash map probe per lookup (i.e. <code>numProbes / numKeyLookups</code>)</p> <p>Note <code>numProbes</code> and <code>numKeyLookups</code> are used in BytesToBytesMap append-only hash map for the number of iteration to look up a single key and the number of all the lookups in total, respectively.</p>
numOutputRows	number of output rows	<p>Number of groups (per partition) that (depending on the number of partitions and the side of ShuffleExchangeExec operator) is the number of groups</p> <ul style="list-style-type: none"> • <code>0</code> for no input with a grouping expression, e.g. <code>spark.range(0).groupBy(\$"id").count.show</code> • <code>1</code> for no grouping expression and no input, e.g. <code>spark.range(0).groupBy().count.show</code> <p>Tip Use different number of elements and partitions in <code>range</code> operator to observe the difference in <code>numOutputRows</code> metric, e.g.</p> <pre> spark. range(0, 10, 1, numPartitions = 1). groupBy(\$"id" % 5 as "gid"). count. show spark. range(0, 10, 1, numPartitions = 5). groupBy(\$"id" % 5 as "gid"). count. show </pre>
peakMemory	peak memory	
spillSize	spill size	

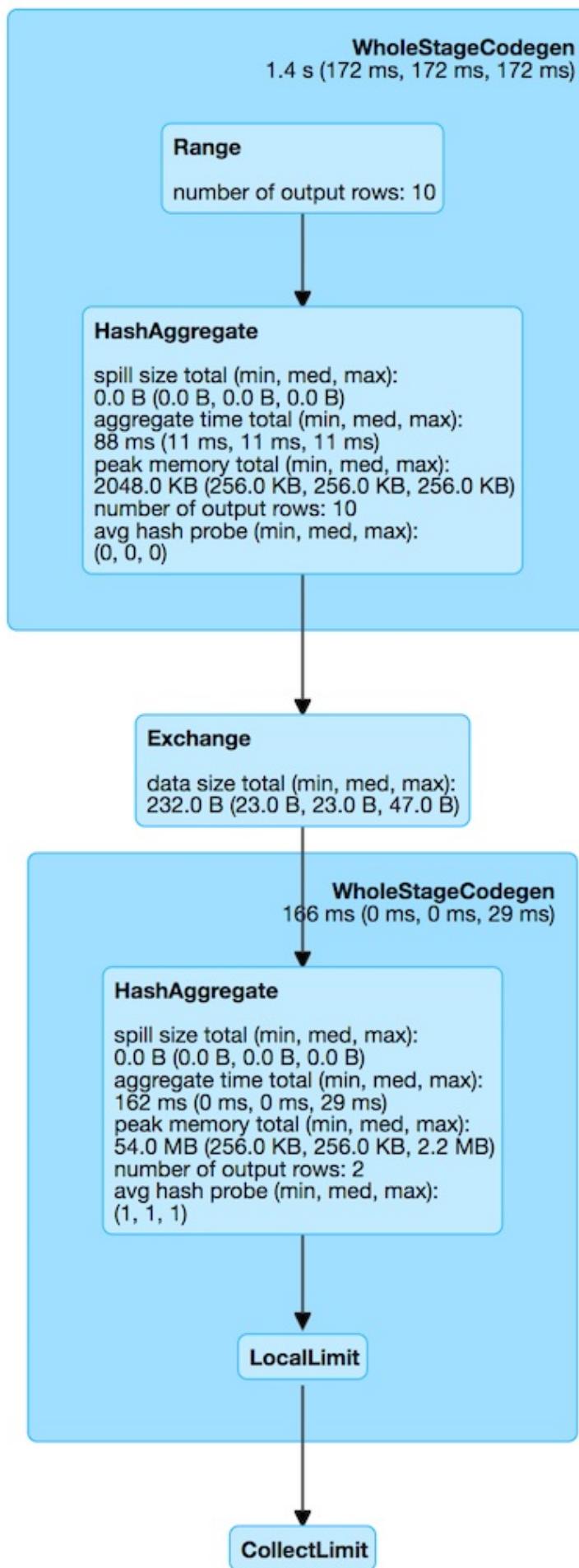


Figure 1. HashAggregateExec in web UI (Details for Query)

Table 2. HashAggregateExec's Properties

Name	Description
output	Output schema for the input NamedExpressions

`requiredChildDistribution` varies per the input required child distribution expressions.

Table 3. HashAggregateExec's Required Child Output Distributions

requiredChildDistributionExpressions	Distribution
Defined, but empty	AllTuples
Non-empty	ClusteredDistribution for exprs
Undefined (None)	UnspecifiedDistribution

`requiredChildDistributionExpressions` is exactly `requiredChildDistributionExpressions` from [AggUtils.createAggregate](#) and is undefined by default.

(No distinct in aggregation) `requiredChildDistributionExpressions` is undefined when `HashAggregateExec` is created for partial aggregations (i.e. `mode` is `Partial` for aggregate expressions).

`requiredChildDistributionExpressions` is defined, but could possibly be empty, when `HashAggregateExec` is created for final aggregations (i.e. `mode` is `Final` for aggregate expressions).

(one distinct in aggregation) `requiredChildDistributionExpressions` is undefined when `HashAggregateExec` is created for partial aggregations (i.e. `mode` is `Partial` for aggregate expressions) with one distinct in aggregation.

`requiredChildDistributionExpressions` is defined, but could possibly be empty, when `HashAggregateExec` is created for partial merge aggregations (i.e. `mode` is `PartialMerge` for aggregate expressions).

FIXME for the following two cases in aggregation with one distinct.

Note

The prefix for variable names for `HashAggregateExec` operators in [CodegenSupport](#)-generated code is `agg`.

Table 4. HashAggregateExec's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
aggregateBufferAttributes	All the AttributeReferences of the AggregateFunctions of the AggregateExpressions
testFallbackStartsAt	Optional pair of numbers for controlled fall-back to a sort-based aggregation when the hash-based approach is unable to acquire enough memory.
declFunctions	DeclarativeAggregate expressions (from the AggregateFunctions of the AggregateExpressions)
bufferSchema	StructType built from the aggregateBufferAttributes
groupingKeySchema	StructType built from the groupingAttributes
groupingAttributes	Attributes of the groupingExpressions
Note	<p>HashAggregateExec uses TungstenAggregationIterator that can (theoretically) switch to a sort-based aggregation when the hash-based approach is unable to acquire enough memory.</p> <p>See testFallbackStartsAt internal property and spark.sql.TungstenAggregate.testFallbackStartsAt Spark property.</p> <p>Search logs for the following INFO message to know whether the switch has happened.</p> <pre>INFO TungstenAggregationIterator: falling back to sort based aggregation.</pre>

finishAggregate Method

```
finishAggregate(
    hashMap: UnsafeFixedWidthAggregationMap,
    sorter: UnsafeKVExternalSorter,
    peakMemory: SQLMetric,
    spillSize: SQLMetric,
    avgHashProbe: SQLMetric): KVIterator[UnsafeRow, UnsafeRow]
```

```
finishAggregate ...FIXME
```

Note	finishAggregate is used exclusively when HashAggregateExec is requested to generate the Java code for doProduceWithKeys .
------	---

Generating Java Source Code for Whole-Stage Consume Path with Grouping Keys — `doConsumeWithKeys` Internal Method

```
doConsumeWithKeys(ctx: CodegenContext, input: Seq[ExprCode]): String
```

`doConsumeWithKeys` ...FIXME

Note

`doConsumeWithKeys` is used exclusively when `HashAggregateExec` is requested to generate the Java code for whole-stage consume path (with named expressions for the grouping keys).

Generating Java Source Code for Whole-Stage Consume Path without Grouping Keys — `doConsumeWithoutKeys` Internal Method

```
doConsumeWithoutKeys(ctx: CodegenContext, input: Seq[ExprCode]): String
```

`doConsumeWithoutKeys` ...FIXME

Note

`doConsumeWithoutKeys` is used exclusively when `HashAggregateExec` is requested to generate the Java code for whole-stage consume path (with no named expressions for the grouping keys).

Generating Java Source Code for Consume Path in Whole-Stage Code Generation — `doConsume` Method

```
doConsume(ctx: CodegenContext, input: Seq[ExprCode], row: ExprCode): String
```

Note

`doConsume` is part of [CodegenSupport Contract](#) to generate the Java source code for [consume path](#) in Whole-Stage Code Generation.

`doConsume` executes `doConsumeWithoutKeys` when no named expressions for the grouping keys were specified for the `HashAggregateExec` or `doConsumeWithKeys` otherwise.

Generating Java Source Code For "produce" Path (In Whole-Stage Code Generation) — `doProduceWithKeys` Internal Method

```
doProduceWithKeys(ctx: CodegenContext): String
```

`doProduceWithKeys` ...FIXME

Note

`doProduceWithKeys` is used exclusively when `HashAggregateExec` physical operator is requested to generate the Java source code for "produce" path in whole-stage code generation (when there are no `groupingExpressions`).

doProduceWithoutKeys Internal Method

```
doProduceWithoutKeys(ctx: CodegenContext): String
```

`doProduceWithoutKeys` ...FIXME

Note

`doProduceWithoutKeys` is used exclusively when `HashAggregateExec` physical operator is requested to generate the Java source code for "produce" path in whole-stage code generation.

generateResultFunction Internal Method

```
generateResultFunction(ctx: CodegenContext): String
```

`generateResultFunction` ...FIXME

Note

`generateResultFunction` is used exclusively when `HashAggregateExec` physical operator is requested to `doProduceWithKeys` (when `HashAggregateExec` physical operator is requested to generate the Java source code for "produce" path in whole-stage code generation)

supportsAggregate Object Method

```
supportsAggregate(aggregateBufferAttributes: Seq[Attribute]): Boolean
```

`supportsAggregate` firstly creates the schema (from the input aggregation buffer attributes) and requests `UnsafeFixedWidthAggregationMap` to `supportsAggregationBufferSchema` (i.e. the schema uses mutable field data types only that have fixed length and can be mutated in place in an `UnsafeRow`).

Note

`supportsAggregate` is used when:

- `AggUtils` is requested to [creates an aggregate physical operator given aggregate expressions](#)
- `HashAggregateExec` physical operator is [created](#) (to assert that the `aggregateBufferAttributes` are supported)

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

`doExecute(): RDD[InternalRow]`

Note

`doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` requests the `child` physical operator to [execute](#) (that triggers physical query planning and generates an `RDD[InternalRow]`) and transforms it by executing the following function on internal rows per partition with index (using `RDD.mapPartitionsWithIndex` that creates another RDD):

1. Records the start execution time (`beforeAgg`)
2. Requests the `Iterator[InternalRow]` (from executing the `child` physical operator) for the next element
 - i. If there is no input (an empty partition), but there are [grouping keys](#) used, `doExecute` simply returns an empty iterator
 - ii. Otherwise, `doExecute` creates a [TungstenAggregationIterator](#) and branches off per whether there are rows to process and the [grouping keys](#).

For empty partitions and no [grouping keys](#), `doExecute` increments the `numOutputRows` metric and requests the `TungstenAggregationIterator` to [create a single UnsafeRow](#) as the only element of the result iterator.

For non-empty partitions or there are [grouping keys](#) used, `doExecute` returns the `TungstenAggregationIterator`.

In the end, `doExecute` calculates the `aggTime` metric and returns an `Iterator[UnsafeRow]` that can be as follows:

- Empty

- A single-element `Iterator[UnsafeRow]` with the [single UnsafeRow](#)
- The [TungstenAggregationIterator](#)

Note The `numOutputRows`, `peakMemory`, `spillSize` and `avgHashProbe` metrics are used exclusively to create the [TungstenAggregationIterator](#).

`doExecute` (by `RDD.mapPartitionsWithIndex` transformation) adds a new `MapPartitionsRDD` to the RDD lineage. Use `RDD.toDebugString` to see the additional `MapPartitionsRDD`.

```

val ids = spark.range(1)
scala> println(ids.queryExecution.toRdd.toDebugString)
(8) MapPartitionsRDD[12] at toRdd at <console>:29 []
|  MapPartitionsRDD[11] at toRdd at <console>:29 []
|  ParallelCollectionRDD[10] at toRdd at <console>:29 []

// Use groupBy that gives HashAggregateExec operator
val q = ids.groupBy('id).count
scala> q.explain
== Physical Plan ==
*(2) HashAggregate(keys=[id#30L], functions=[count(1)])
+- Exchange hashpartitioning(id#30L, 200)
   +- *(1) HashAggregate(keys=[id#30L], functions=[partial_count(1)])
      +- *(1) Range (0, 1, step=1, splits=8)

val rdd = q.queryExecution.toRdd
scala> println(rdd.toDebugString)
(200) MapPartitionsRDD[18] at toRdd at <console>:28 []
|  ShuffledRowRDD[17] at toRdd at <console>:28 []
+- (8) MapPartitionsRDD[16] at toRdd at <console>:28 []
|  MapPartitionsRDD[15] at toRdd at <console>:28 []
|  MapPartitionsRDD[14] at toRdd at <console>:28 []
|  ParallelCollectionRDD[13] at toRdd at <console>:28 []

```

Note

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

```
doProduce(ctx: CodegenContext): String
```

Note

`doProduce` is part of [CodegenSupport Contract](#) to generate the Java source code for [produce path](#) in Whole-Stage Code Generation.

`doProduce` executes `doProduceWithoutKeys` when no [named expressions](#) for the grouping `keys` were specified for the `HashAggregateExec` or `doProduceWithKeys` otherwise.

Creating HashAggregateExec Instance

`HashAggregateExec` takes the following when created:

- Required child distribution [expressions](#)

- Named expressions for grouping keys
- AggregateExpressions
- Aggregate attributes
- Initial input buffer offset
- Output named expressions
- Child physical plan

`HashAggregateExec` initializes the internal registries and counters.

Creating UnsafeFixedWidthAggregationMap Instance

— `createHashMap` Method

```
createHashMap(): UnsafeFixedWidthAggregationMap
```

`createHashMap` creates a `UnsafeFixedWidthAggregationMap` (with the empty aggregation buffer, the `bufferSchema`, the `groupingKeySchema`, the current `TaskMemoryManager` , `1024 * 16` initial capacity and the page size of the `TaskMemoryManager`)

Note

`createHashMap` is used exclusively when `HashAggregateExec` physical operator is requested to generate the Java source code for "produce" path (in Whole-Stage Code Generation).

HiveTableScanExec Leaf Physical Operator

`HiveTableScanExec` is a [leaf physical operator](#) that represents a [HiveTableRelation](#) logical operator at execution time.

`HiveTableScanExec` is [created](#) exclusively when [HiveTableScans](#) execution planning strategy plans a [HiveTableRelation](#) logical operator (i.e. is executed on a logical query plan with a [HiveTableRelation](#) logical operator).

Table 1. HiveTableScanExec's Performance Metrics

Key	Name (in web UI)	Description
<code>numOutputRows</code>	number of output rows	

Table 2. HiveTableScanExec's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>hiveQlTable</code>	Hive's <code>Table</code> metadata (converted from the CatalogTable of the HiveTableRelation) Used when <code>HiveTableScanExec</code> is requested for the <code>tableDesc</code> , <code>rawPartitions</code> and is executed
<code>rawPartitions</code>	
<code>tableDesc</code>	Hive's <code>TableDesc</code>

Creating HiveTableScanExec Instance

`HiveTableScanExec` takes the following when created:

- Requested [attributes](#)
- [HiveTableRelation](#)
- Partition pruning predicate [expression](#)
- [SparkSession](#)

`HiveTableScanExec` initializes the [internal registries and counters](#).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	<p><code>doExecute</code> is part of SparkPlan Contract to generate the runtime representation of a structured query as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).</p>
------	--

`doExecute` ...FIXME

InMemoryTableScanExec Leaf Physical Operator

`InMemoryTableScanExec` is a [leaf physical operator](#) that represents an [InMemoryRelation](#) logical operator at execution time.

`InMemoryTableScanExec` is [created](#) exclusively when `InMemoryScans` execution planning strategy is [executed](#) and finds an [InMemoryRelation](#) logical operator in a logical query plan.

`InMemoryTableScanExec` takes the following to be created:

- [Attribute](#) expressions
- [Predicate](#) expressions
- [InMemoryRelation](#) logical operator

`InMemoryTableScanExec` is a [ColumnarBatchScan](#) that [supports batch decoding](#) (when [created](#) for a [DataSourceReader](#) that supports it, i.e. the `DataSourceReader` is a [SupportsScanColumnarBatch](#) with the `enableBatchRead` flag enabled).

`InMemoryTableScanExec` supports [partition batch pruning](#) (only when `spark.sql.inMemoryColumnarStorage.partitionPruning` internal configuration property is enabled which is so by default).

```
// Sample DataFrames
val tokens = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "InMemoryTableScanExec")
).toDF("id", "token")
val ids = spark.range(10)

// Cache DataFrames
tokens.cache
ids.cache

val q = tokens.join(ids, Seq("id"), "outer")
scala> q.explain
== Physical Plan ==
*Project [coalesce(cast(id#5 as bigint), id#10L) AS id#33L, token#6]
+- SortMergeJoin [cast(id#5 as bigint)], [id#10L], FullOuter
  :- *Sort [cast(id#5 as bigint) ASC NULLS FIRST], false, 0
  :  +- Exchange hashpartitioning(cast(id#5 as bigint), 200)
  :    +- InMemoryTableScan [id#5, token#6]
  :         +- InMemoryRelation [id#5, token#6], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
  :             +- LocalTableScan [id#5, token#6]
  +- *Sort [id#10L ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(id#10L, 200)
      +- InMemoryTableScan [id#10L]
         +- InMemoryRelation [id#10L], true, 10000, StorageLevel(disk, memory, d
eserialized, 1 replicas)
         +- *Range (0, 10, step=1, splits=8)
```

```
val q = spark.range(4).cache
val plan = q.queryExecution.executedPlan
import org.apache.spark.sql.execution.columnar.InMemoryTableScanExec
val inmemoryScan = plan.collectFirst { case exec: InMemoryTableScanExec => exec }.get
assert(inmemoryScan.supportCodegen == inmemoryScan.supportsBatch)
```

Table 1. InMemoryTableScanExec's Performance Metrics

Key	Name (in web UI)	Description
numOutputRows	number of output rows	

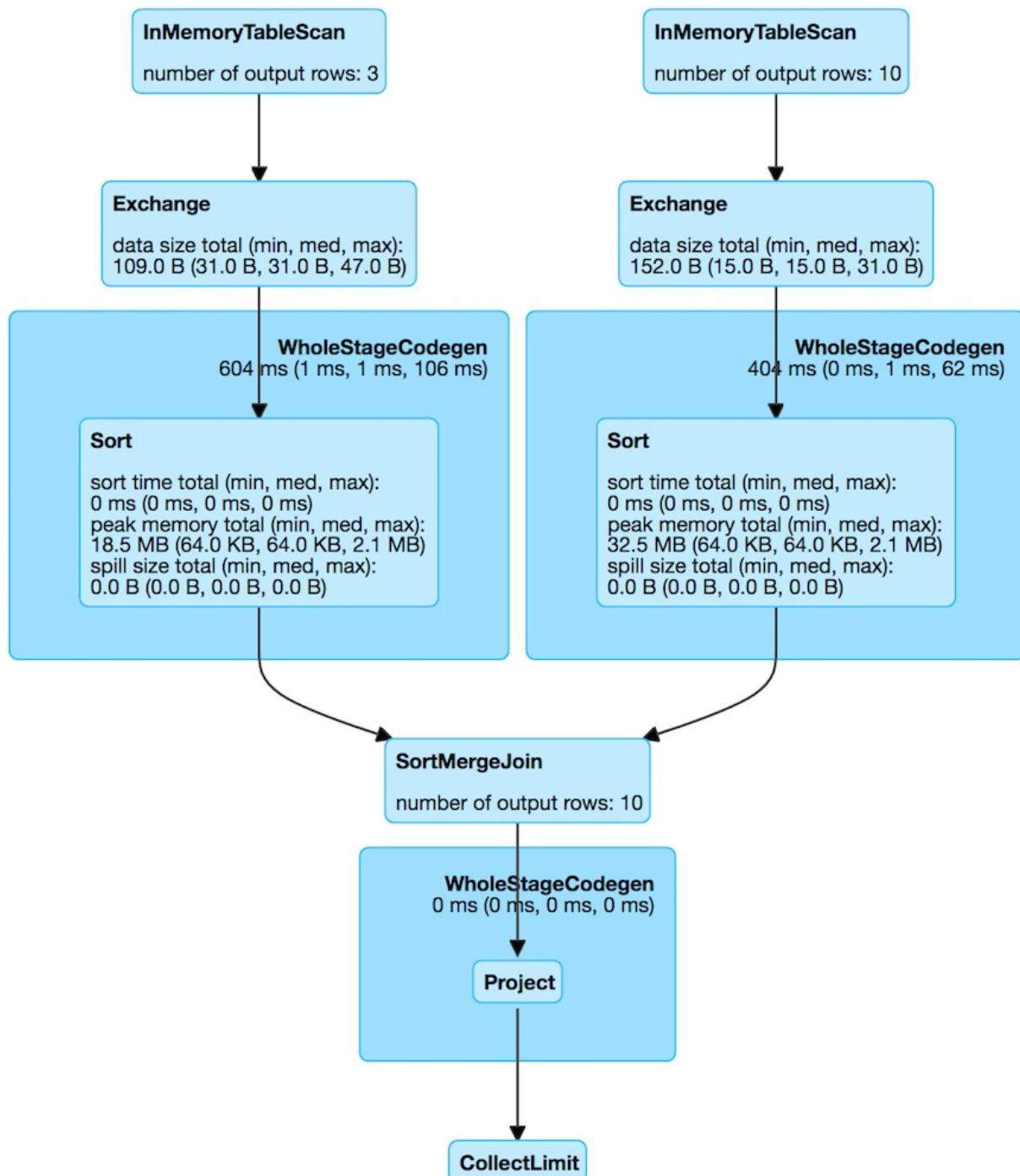


Figure 1. InMemoryTableScanExec in web UI (Details for Query)

InMemoryTableScanExec supports Java code generation only if batch decoding is enabled.

InMemoryTableScanExec gives the single `inputRDD` as the only RDD of internal rows (when `wholeStageCodegenExec` physical operator is executed).

InMemoryTableScanExec uses `spark.sql.inMemoryTableScanStatistics.enable` flag (default: `false`) to enable accumulators (that seems to be exclusively for testing purposes).

Table 2. InMemoryTableScanExec's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
columnarBatchSchema	<p>Schema of a columnar batch</p> <p>Used exclusively when <code>InMemoryTableScanExec</code> is requested to createAndDecompressColumn.</p>
stats	<p>PartitionStatistics of the <code>InMemoryRelation</code></p> <p>Used when <code>InMemoryTableScanExec</code> is requested for partitionFilters, partition batch pruning and statsFor.</p>

vectorTypes Method

```
vectorTypes: Option[Seq[String]]
```

Note

`vectorTypes` is part of [ColumnarBatchScan Contract](#) to...FIXME.

`vectorTypes` uses `spark.sql.columnVector.offheap.enabled` internal configuration property to select the name of the concrete column vector, i.e. [OnHeapColumnVector](#) or [OffHeapColumnVector](#) when the property is off or on, respectively.

`vectorTypes` gives as many column vectors as the [attribute expressions](#).

supportsBatch Property

```
supportsBatch: Boolean
```

Note

`supportsBatch` is part of [ColumnarBatchScan Contract](#) to control whether the physical operator supports vectorized decoding or not.

`supportsBatch` is enabled when all of the following holds:

1. `spark.sql.inMemoryColumnarStorage.enableVectorizedReader` configuration property is enabled (default: `true`)
2. The [output schema](#) of the `InMemoryRelation` uses primitive data types only, i.e. `BooleanType`, `ByteType`, `ShortType`, `IntegerType`, `LongType`, `FloatType`, `DoubleType`
3. The number of nested fields in the output schema of the `InMemoryRelation` is at most `spark.sql_codegen.maxFields` internal configuration property

partitionFilters Property

```
partitionFilters: Seq[Expression]
```

Note

`partitionFilters` is a Scala lazy value which is computed once when accessed and cached afterwards.

`partitionFilters ...FIXME`

Note

`partitionFilters` is used when...FIXME

Applying Partition Batch Pruning to Cached Column Buffers (Creating MapPartitionsRDD of Filtered CachedBatches) — `filteredCachedBatches` Internal Method

```
filteredCachedBatches(): RDD[CachedBatch]
```

`filteredCachedBatches` requests [PartitionStatistics](#) for the output schema and [InMemoryRelation](#) for [cached column buffers](#) (as a `RDD[CachedBatch]`).

`filteredCachedBatches` takes the cached column buffers (as a `RDD[CachedBatch]`) and transforms the RDD per partition with index (i.e. `RDD.mapPartitionsWithIndexInternal`) as follows:

1. Creates a partition filter as a new [GenPredicate](#) for the `partitionFilters` expressions (concatenated together using `And` binary operator and the schema)
2. Requests the generated partition filter `Predicate` to `initialize`
3. Uses [spark.sql.inMemoryColumnarStorage.partitionPruning](#) internal configuration property to enable **partition batch pruning** and filtering out (skipping) `CachedBatches` in a partition based on column stats and the generated partition filter `Predicate`

Note

If [spark.sql.inMemoryColumnarStorage.partitionPruning](#) internal configuration property is disabled (i.e. `false`), `filteredCachedBatches` does nothing and simply passes all `CachedBatch` elements along.

Note

[spark.sql.inMemoryColumnarStorage.partitionPruning](#) internal configuration property is enabled by default.

Note

`filteredCachedBatches` is used exclusively when `InMemoryTableScanExec` is requested for the `inputRDD` internal property.

statsFor Internal Method

```
statsFor(a: Attribute)
```

`statsFor ...FIXME`

Note	<code>statsFor</code> is used when...FIXME
------	--

createAndDecompressColumn Internal Method

```
createAndDecompressColumn(cachedColumnarBatch: CachedBatch): ColumnarBatch
```

`createAndDecompressColumn` takes the number of rows in the input `CachedBatch`.

`createAndDecompressColumn` requests [OffHeapColumnVector](#) or [OnHeapColumnVector](#) to allocate column vectors (with the number of rows and [columnarBatchSchema](#)) per the [spark.sql.columnVector.offheap.enabled](#) internal configuration flag, i.e. `true` or `false`, respectively.

Note	<code>spark.sql.columnVector.offheap.enabled</code> internal configuration flag is disabled by default which means that OnHeapColumnVector is used.
------	---

`createAndDecompressColumn` creates a [ColumnarBatch](#) for the allocated column vectors (as an array of `ColumnVector`).

`createAndDecompressColumn` sets the number of rows in the columnar batch.

For every [Attribute](#) `createAndDecompressColumn` requests `ColumnAccessor` to `decompress` the column.

`createAndDecompressColumn` registers a callback to be executed on a task completion that will close the `ColumnarBatch`.

In the end, `createAndDecompressColumn` returns the `ColumnarBatch`.

Note	<code>createAndDecompressColumn</code> is used exclusively when <code>InMemoryTableScanExec</code> is requested for the input RDD of internal rows .
------	--

Creating Input RDD of Internal Rows — `inputRDD` Internal Property

```
inputRDD: RDD[InternalRow]
```

Note

`inputRDD` is a Scala lazy value which is computed once when accessed and cached afterwards.

`inputRDD` firstly applies partition batch pruning to cached column buffers (and creates a filtered cached batches as a `RDD[CachedBatch]`).

With `supportsBatch` flag on, `inputRDD` finishes with a new `MapPartitionsRDD` (using `RDD.map`) by `createAndDecompressColumn` on all cached columnar batches.

Caution

Show examples of `supportsBatch` enabled and disabled

```
// Demo: A MapPartitionsRDD in the RDD lineage
val q = spark.range(4).cache
val plan = q.queryExecution.executedPlan
import org.apache.spark.sql.execution.columnar.InMemoryTableScanExec
val inmemoryScan = plan.collectFirst { case exec: InMemoryTableScanExec => exec }.get

// supportsBatch flag is on since the schema is a single column of longs
assert(inmemoryScan.supportsBatch)

val rdd = inmemoryScan.inputRDDs.head
scala> rdd.toDebugString
res2: String =
(8) MapPartitionsRDD[5] at inputRDDs at <console>:27 []
| MapPartitionsRDD[4] at inputRDDs at <console>:27 []
| *(1) Range (0, 4, step=1, splits=8)
MapPartitionsRDD[3] at cache at <console>:23 []
| MapPartitionsRDD[2] at cache at <console>:23 []
| MapPartitionsRDD[1] at cache at <console>:23 []
| ParallelCollectionRDD[0] at cache at <console>:23 []
```

With `supportsBatch` flag off, `inputRDD` firstly applies partition batch pruning to cached column buffers (and creates a filtered cached batches as a `RDD[CachedBatch]`).

Note

Indeed. `inputRDD` applies partition batch pruning to cached column buffers (and creates a filtered cached batches as a `RDD[CachedBatch]`) twice which seems unnecessary.

In the end, `inputRDD` creates a new `MapPartitionsRDD` (using `RDD.map`) with a `ColumnarIterator` applied to all cached columnar batches that is created as follows:

1. For every `CachedBatch` in the partition iterator adds the total number of rows in the batch to `numOutputRows` SQL metric
2. Requests `GenerateColumnAccessor` to generate the Java code for a `ColumnarIterator` to perform expression evaluation for the given `column types`.
3. Requests `ColumnarIterator` to initialize

```
// Demo: A MapPartitionsRDD in the RDD lineage (supportsBatch flag off)
import java.sql.Date
import java.time.LocalDate
val q = Seq(LocalDate.now()).toDF("date").cache
val plan = q.queryExecution.executedPlan

import org.apache.spark.sql.execution.columnar.InMemoryTableScanExec
val inmemoryScan = plan.collectFirst { case exec: InMemoryTableScanExec => exec }.get

// supportsBatch flag is off since the schema uses java.sql.Date
assert(inmemoryScan.supportsBatch == false)

val rdd = inmemoryScan.inputRDDs.head
scala> rdd.toDebugString
res2: String =
(1) MapPartitionsRDD[12] at inputRDDs at <console>:28 []
 | MapPartitionsRDD[11] at inputRDDs at <console>:28 []
 | LocalTableScan [date#15]
MapPartitionsRDD[9] at cache at <console>:25 []
 | MapPartitionsRDD[8] at cache at <console>:25 []
 | ParallelCollectionRDD[7] at cache at <console>:25 []
```

Note

`inputRDD` is used when `InMemoryTableScanExec` is requested for the `input RDDs` and to `execute`.

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` branches off per `supportsBatch` flag.

With `supportsBatch` flag on, `doExecute` creates a [WholeStageCodegenExec](#) (with the `InMemoryTableScanExec` physical operator as the `child` and `codegenStageId` as `0`) and requests it to `execute`.

Otherwise, when `supportsBatch` flag is off, `doExecute` simply gives the `input RDD` of internal rows.

buildFilter Property

```
buildFilter: PartialFunction[Expression, Expression]
```

Note `buildFilter` is a Scala lazy value which is computed once when accessed and cached afterwards.

`buildFilter` is a Scala [PartialFunction](#) that accepts an [Expression](#) and produces an [Expression](#), i.e. `PartialFunction[Expression, Expression]`.

Table 3. `buildFilter`'s Expressions

Input Expression	Description
<code>And</code>	
<code>Or</code>	
<code>EqualTo</code>	
<code>EqualNullSafe</code>	
<code>LessThan</code>	
<code>LessThanOrEqual</code>	
<code>GreaterThan</code>	
<code>GreaterThanOrEqual</code>	
<code>IsNull</code>	
<code>IsNotNull</code>	
<code>In</code> with a non-empty <code>list</code> of Literal expressions	<p>For every <code>Literal</code> expression in the expression list, <code>buildFilter</code> creates an <code>And</code> expression with the lower and upper bounds of the partition statistics for the attribute and the <code>Literal</code>.</p> <p>In the end, <code>buildFilter</code> joins the <code>And</code> expressions with <code>or</code> expressions.</p>

Note `buildFilter` is used exclusively when `InMemoryTableScanExec` is requested for [partitionFilters](#).

innerChildren Method

```
innerChildren: Seq[QueryPlan[_]]
```

Note

`innerChildren` is part of [QueryPlan Contract](#) to...FIXME.

`innerChildren` ...FIXME

LocalTableScanExec Leaf Physical Operator

`LocalTableScanExec` is a [leaf physical operator](#) (i.e. no [children](#)) and `producedAttributes` being `outputSet`.

`LocalTableScanExec` is [created](#) when [BasicOperators](#) execution planning strategy resolves [LocalRelation](#) and Spark Structured Streaming's `MemoryPlan` logical operators.

Tip	Read on MemoryPlan logical operator in the Spark Structured Streaming gitbook.
-----	--

```

val names = Seq("Jacek", "Agata").toDF("name")
val optimizedPlan = names.queryExecution.optimizedPlan

scala> println(optimizedPlan.numberedTreeString)
00 LocalRelation [name#9]

// Physical plan with LocalTableScanExec operator (shown as LocalTableScan)
scala> names.explain
== Physical Plan ==
LocalTableScan [name#9]

// Going fairly low-level...you've been warned

val plan = names.queryExecution.executedPlan
import org.apache.spark.sql.execution.LocalTableScanExec
val ltse = plan.asInstanceOf[LocalTableScanExec]

val ltseRDD = ltse.execute()
scala> :type ltseRDD
org.apache.spark.rdd.RDD[org.apache.spark.sql.catalyst.InternalRow]

scala> println(ltseRDD.toDebugString)
(2) MapPartitionsRDD[1] at execute at <console>:30 []
| ParallelCollectionRDD[0] at execute at <console>:30 []

// no computation on the source dataset has really occurred yet
// Let's trigger a RDD action
scala> ltseRDD.first
res6: org.apache.spark.sql.catalyst.InternalRow = [0,1000000005,6b6563614a]

// Low-level "show"
scala> ltseRDD.foreach(println)
[0,1000000005,6b6563614a]
[0,1000000005,6174616741]

// High-level show
scala> names.show
+---+
| name|
+---+
| Jacek|
| Agata|
+---+

```

Table 1. LocalTableScanExec's Performance Metrics

Key	Name (in web UI)	Description
numOutputRows	number of output rows	

It appears that when no Spark job is used to execute a LocalTableScanExec the numOutputRows metric is not displayed in the web UI.

Note

```
val names = Seq("Jacek", "Agata").toDF("name")

// The following query gives no numOutputRows metric in web UI's Details for Query (SQL tab)
scala> names.show
+---+
| name|
+---+
|Jacek|
|Agata|
+---+

// The query gives numOutputRows metric in web UI's Details for Query (SQL tab)
scala> names.groupBy(length($"name")).count.show
+-----+---+
|length(name)|count|
+-----+---+
|           5|     2|
+-----+---+


// The (type-preserving) query does also give numOutputRows metric in web UI's Details for Query (SQL tab)
scala> names.as[String].map(_.toUpperCase).show
+---+
|value|
+---+
|JACEK|
|AGATA|
+---+
```

When executed, LocalTableScanExec ...FIXME

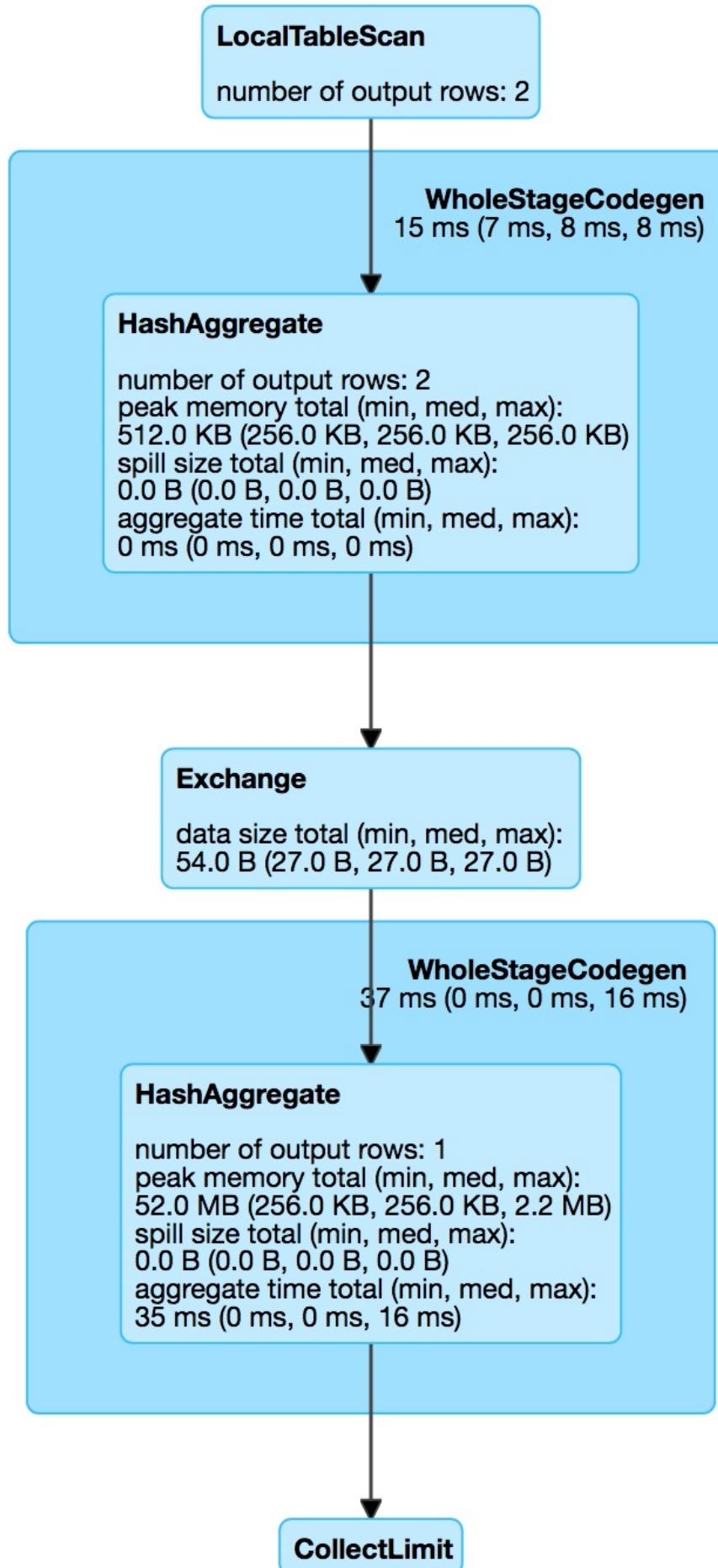


Figure 1. LocalTableScanExec in web UI (Details for Query)

Table 2. LocalTableScanExec's Internal Properties

Name	Description
unsafeRows	Internal binary rows for...FIXME
numParallelism	
rdd	

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	<code>doExecute</code> is part of SparkPlan Contract to generate the runtime representation of a structured query as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	---

`doExecute` ...FIXME

Creating LocalTableScanExec Instance

`LocalTableScanExec` takes the following when created:

- Output schema [attributes](#)
- [Internal binary rows](#)

MapElementsExec

MapElementsExec is...FIXME

ObjectHashAggregateExec Aggregate Physical Operator

`ObjectHashAggregateExec` is a [unary physical operator](#) (i.e. with one [child](#) physical operator) that is [created](#) (indirectly through [AggUtils.createAggregate](#)) when:

- ...FIXME

```
// ObjectHashAggregateExec selected due to:
// 1. spark.sql.execution.useObjectHashAggregateExec internal flag is enabled
scala> val objectHashEnabled = spark.conf.get("spark.sql.execution.useObjectHashAggregateExec")
objectHashEnabled: String = true

// 2. The following data types are used in aggregateBufferAttributes
// BinaryType
// StringType
// ArrayType
// MapType
// ObjectType
// StructType
val dataset = Seq(
  (0, Seq.empty[Int]),
  (1, Seq(1, 1)),
  (2, Seq(2, 2))).toDF("id", "nums")
import org.apache.spark.sql.functions.size
val q = dataset.
  groupBy(size($"nums") as "group"). // <-- size over array
  agg(collect_list("id") as "ids")
scala> q.explain
== Physical Plan ==
ObjectHashAggregate(keys=[size(nums#113)#127], functions=[collect_list(id#112, 0, 0)])
+- Exchange hashpartitioning(size(nums#113)#127, 200)
   +- ObjectHashAggregate(keys=[size(nums#113) AS size(nums#113)#127], functions=[partial_collect_list(id#112, 0, 0)])
      +- LocalTableScan [id#112, nums#113]

scala> println(q.queryExecution.sparkPlan.numberedTreeString)
00 ObjectHashAggregate(keys=[size(nums#113)#130], functions=[collect_list(id#112, 0, 0)], output=[group#117, ids#122])
01 +- ObjectHashAggregate(keys=[size(nums#113) AS size(nums#113)#130], functions=[partial_collect_list(id#112, 0, 0)], output=[size(nums#113)#130, buf#132])
02   +- LocalTableScan [id#112, nums#113]

// Going low level...watch your steps :)

// copied from HashAggregateExec as it is the preferred aggregate physical operator
// and HashAggregateExec is checked first
```

```

// When the check fails, ObjectHashAggregateExec is then checked
import q.execution.optimizedPlan
import org.apache.spark.sql.catalyst.plans.logical.Aggregate
val aggLog = optimizedPlan.asInstanceOf[Aggregate]
import org.apache.spark.sql.catalyst.planning.PhysicalAggregation
import org.apache.spark.sql.catalyst.expressions.aggregate.AggregateExpression
val aggregateExpressions: Seq[AggregateExpression] = PhysicalAggregation.unapply(aggLog).get._2
val aggregateBufferAttributes = aggregateExpressions.
  flatMap(_.aggregateFunction.aggBufferAttributes)
import org.apache.spark.sql.execution.aggregate.HashAggregateExec
// that's one of the reasons why ObjectHashAggregateExec was selected
// HashAggregateExec did not meet the requirements
scala> val useHash = HashAggregateExec.supportsAggregate(aggregateBufferAttributes)
useHash: Boolean = true

// collect_list aggregate function uses CollectList TypedImperativeAggregate under the
// covers
import org.apache.spark.sql.execution.aggregate.ObjectHashAggregateExec
scala> val useObjectHash = ObjectHashAggregateExec.supportsAggregate(aggregateExpressions)
useObjectHash: Boolean = true

val aggExec = q.execution.sparkPlan.children.head.asInstanceOf[ObjectHashAggregateExec]
scala> println(aggExec.aggregateExpressions.head.numberedTreeString)
00 partial_collect_list(id#112, 0, 0)
01 +- collect_list(id#112, 0, 0)
02   +- id#112: int

```

Table 1. ObjectHashAggregateExec's Performance Metrics

Key	Name (in web UI)	Description
numOutputRows	number of output rows	

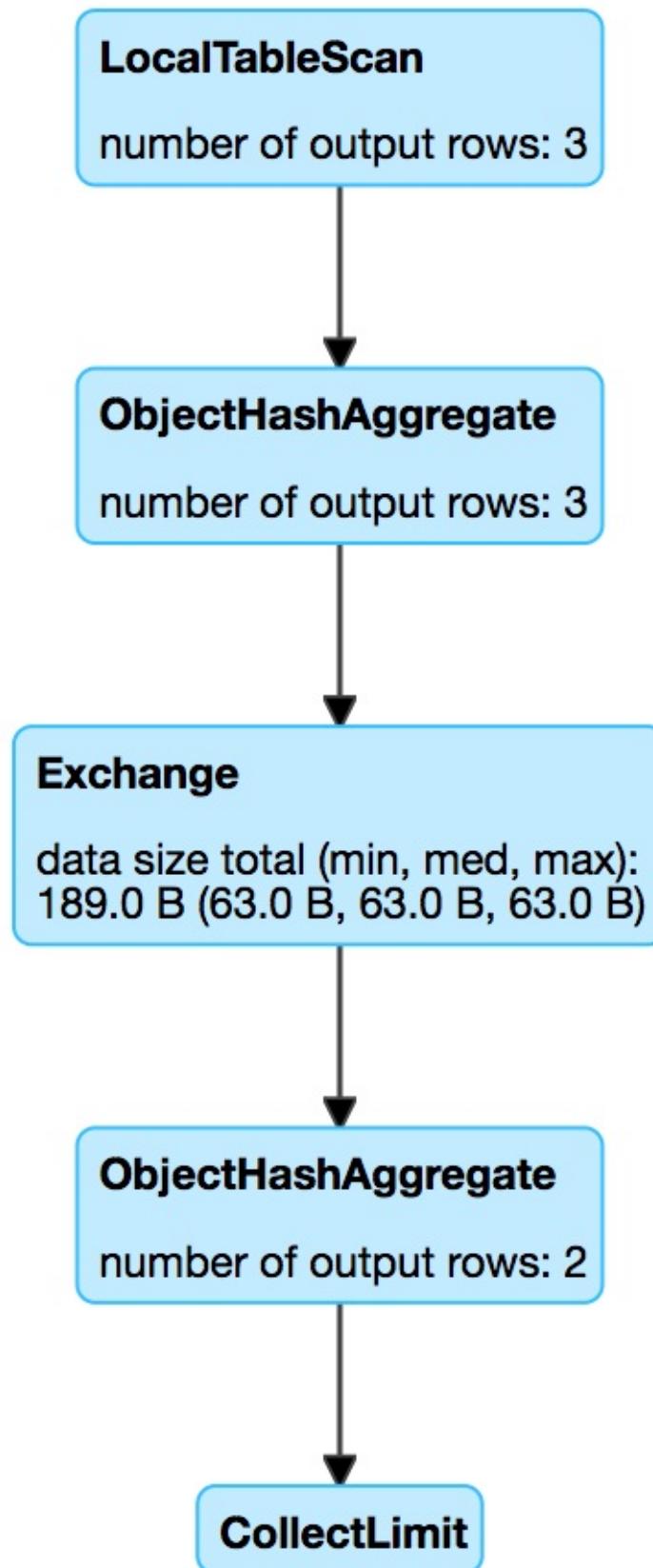


Figure 1. ObjectHashAggregateExec in web UI (Details for Query)

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

```
doExecute(): RDD[InternalRow]
```

Note `doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` ...FIXME

supportsAggregate Method

```
supportsAggregate(aggregateExpressions: Seq[AggregateExpression]): Boolean
```

`supportsAggregate` is enabled (i.e. returns `true`) if there is at least one [TypedImperativeAggregate](#) aggregate function in the input `aggregateExpressions` [aggregate expressions](#).

Note `supportsAggregate` is used exclusively when `AggUtils` is requested to [create an aggregate physical operator given aggregate expressions](#).

Creating ObjectHashAggregateExec Instance

`ObjectHashAggregateExec` takes the following when created:

- Required child distribution [expressions](#)
- Grouping [named expressions](#)
- [Aggregate expressions](#)
- Aggregate [attributes](#)
- Initial input buffer offset
- Output [named expressions](#)
- Child [physical plan](#)

ObjectProducerExec — Physical Operators With Single Object Output

`ObjectProducerExec` is the [extension](#) of the `SparkPlan` contract for [physical operators](#) that produce a single [object](#).

Table 1. ObjectProducerExec Contract (Abstract Methods Only)

Method	Description
<code>outputObjAttr</code>	<code>outputObjAttr: Attribute</code> Used when...FIXME

Table 2. ObjectProducerExecs

ObjectProducerExec	Description
<code>CoGroupExec</code>	
<code>DeserializeToObjectExec</code>	
<code>ExternalRDDScanExec</code>	
<code>FlatMapGroupsInRExec</code>	
<code>FlatMapGroupsWithStateExec</code>	
<code>MapElementsExec</code>	
<code>MapGroupsExec</code>	
<code>MapPartitionsExec</code>	

ProjectExec Unary Physical Operator

`ProjectExec` is a [unary physical operator](#) (i.e. with one `child` physical operator) that...FIXME

`ProjectExec` supports [Java code generation](#) (aka `codegen`).

`ProjectExec` is [created](#) when:

- `InMemoryScans` and `HiveTableScans` execution planning strategies are executed (and request `SparkPlanner` to [pruneFilterProject](#))
- `BasicOperators` execution planning strategy is requested to [resolve a Project logical operator](#)
- `DataSourceStrategy` execution planning strategy is requested to [creates a RowDataSourceScanExec](#)
- `FileSourceStrategy` execution planning strategy is requested to [plan a LogicalRelation with a HadoopFsRelation](#)
- `ExtractPythonUDFs` physical optimization is requested to [optimize a physical query plan](#) (and [extracts Python UDFs](#))

	<p>The following is the order of applying the above execution planning strategies to logical query plans when <code>SparkPlanner</code> or Hive-specific SparkPlanner are requested to plan a logical query plan into one or more physical query plans:</p> <ol style="list-style-type: none"> 1. <code>HiveTableScans</code> 2. <code>FileSourceStrategy</code> 3. <code>DataSourceStrategy</code> 4. <code>InMemoryScans</code> 5. <code>BasicOperators</code>
--	---

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

	<p><code>doExecute</code> is part of SparkPlan Contract to generate the runtime representation of a structured query as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).</p>
--	--

`doExecute` requests the input `child physical plan` to produce an `RDD of internal rows` and applies a `calculation over indexed partitions` (using `RDD.mapPartitionsWithIndexInternal`).

`RDD.mapPartitionsWithIndexInternal`

```
mapPartitionsWithIndexInternal[U](
  f: (Int, Iterator[T]) => Iterator[U],
  preservesPartitioning: Boolean = false)
```

Inside `doExecute (RDD.mapPartitionsWithIndexInternal)`

Inside the function (that is part of `RDD.mapPartitionsWithIndexInternal`), `doExecute` creates an `UnsafeProjection` with the following:

1. `Named expressions`
2. `Output` of the `child` physical operator as the input schema
3. `subexpressionEliminationEnabled` flag

`doExecute` requests the `UnsafeProjection` to `initialize` and maps over the internal rows (of a partition) using the projection.

Creating ProjectExec Instance

`ProjectExec` takes the following when created:

- `NamedExpressions` for the projection
- Child `physical operator`

Generating Java Source Code for Consume Path in Whole-Stage Code Generation — `doConsume` Method

```
doConsume(ctx: CodegenContext, input: Seq[ExprCode], row: ExprCode): String
```

Note

`doConsume` is part of `CodegenSupport Contract` to generate the Java source code for `consume path` in Whole-Stage Code Generation.

`doConsume ...FIXME`

RangeExec Leaf Physical Operator

RangeExec is a [leaf physical operator](#) that...FIXME

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

```
doProduce(ctx: CodegenContext): String
```

Note

`doProduce` is part of [CodegenSupport Contract](#) to generate the Java source code for [produce path](#) in Whole-Stage Code Generation.

doProduce ...FIXME

RDDScanExec Leaf Physical Operator

RDDScanExec is a [leaf physical operator](#) that...FIXME

ReusedExchangeExec Leaf Physical Operator

ReusedExchangeExec is a [leaf physical operator](#) that...FIXME

RowDataSourceScanExec Leaf Physical Operator

`RowDataSourceScanExec` is a [DataSourceScanExec](#) (and so indirectly a [leaf physical operator](#)) for scanning data from a [BaseRelation](#).

`RowDataSourceScanExec` is [created](#) to represent a [LogicalRelation](#) with the following scan types when `DataSourceStrategy` execution planning strategy is [executed](#):

- `CatalystScan`, `PrunedFilteredScan`, `PrunedScan` (indirectly when `DataSourceStrategy` is requested to [pruneFilterProjectRaw](#))
- `TableScan`

`RowDataSourceScanExec` marks the [filters](#) that are included in the [handledFilters](#) with `*` (star) in the [metadata](#) that is used for a simple text representation.

```
// DEMO RowDataSourceScanExec with a simple text representation with stars
```

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

```
doProduce(ctx: CodegenContext): String
```

Note

`doProduce` is part of [CodegenSupport Contract](#) to generate the Java source code for [produce path](#) in Whole-Stage Code Generation.

`doProduce` ...FIXME

Creating RowDataSourceScanExec Instance

`RowDataSourceScanExec` takes the following when created:

- Output schema [attributes](#)
- Indices of required columns
- [Filter predicates](#)
- Handled [filter predicates](#)
- RDD of [internal binary rows](#)

- [BaseRelation](#)
- [TableIdentifier](#)

Note The input [filter predicates](#) and [handled filters predicates](#) are used exclusively for the [metadata](#) property that is part of [DataSourceScanExec Contract](#) to describe a scan for a [simple text representation](#) (in a query plan tree).

metadata Property

```
metadata: Map[String, String]
```

Note [metadata](#) is part of [DataSourceScanExec Contract](#) to describe a scan for a [simple text representation](#) (in a query plan tree).

[metadata](#) marks the [filter predicates](#) that are included in the [handled filters predicates](#) with `*` (star).

Note Filter predicates with `*` (star) are to denote filters that are pushed down to a relation (aka *data source*).

In the end, [metadata](#) creates the following mapping:

1. **ReadSchema** with the [output](#) converted to [catalog](#) representation
2. **PushedFilters** with the marked and unmarked [filter predicates](#)

SampleExec

SampleExec is...FIXME

ShuffleExchangeExec Unary Physical Operator

`ShuffleExchangeExec` is a concrete [Exchange](#) unary physical operator that is used to [perform a shuffle](#).

`ShuffleExchangeExec` corresponds to [Repartition](#) (with shuffle enabled) and [RepartitionByExpression](#) logical operators at execution time (as resolved by [BasicOperators](#) execution planning strategy).

Note	<code>ShuffleExchangeExec</code> presents itself as Exchange in physical query plans.
------	--

```
// Uses Repartition logical operator
// ShuffleExchangeExec with RoundRobinPartitioning
val q1 = spark.range(6).repartition(2)
scala> q1.explain
== Physical Plan ==
Exchange RoundRobinPartitioning(2)
+- *Range (0, 6, step=1, splits=Some(8))

// Uses RepartitionByExpression logical operator
// ShuffleExchangeExec with HashPartitioning
val q2 = spark.range(6).repartition(2, 'id % 2)
scala> q2.explain
== Physical Plan ==
Exchange hashpartitioning((id#38L % 2), 2)
+- *Range (0, 6, step=1, splits=Some(8))
```

`ShuffleExchangeExec` takes the following to be created:

- [Partitioning](#)
- Child [physical operator](#)
- Optional [ExchangeCoordinator](#)

The optional [ExchangeCoordinator](#) is defined only for [Adaptive Query Execution](#) (when [EnsureRequirements](#) physical query optimization is [executed](#)).

When requested for [nodeName](#), `ShuffleExchangeExec` gives **Exchange** prefix possibly followed by **(coordinator id: [coordinator-hash-code])** per the optional [ExchangeCoordinator](#).

When requested for the [output data partitioning requirements](#), `ShuffleExchangeExec` simply returns the [Partitioning](#).

When requested to [prepare for execution](#), `ShuffleExchangeExec` registers itself with the optional [ExchangeCoordinator](#) if defined.

Performance Metrics

Table 1. ShuffleExchangeExec's Performance Metrics

Key	Name (in web UI)	Description
dataSize	data size	

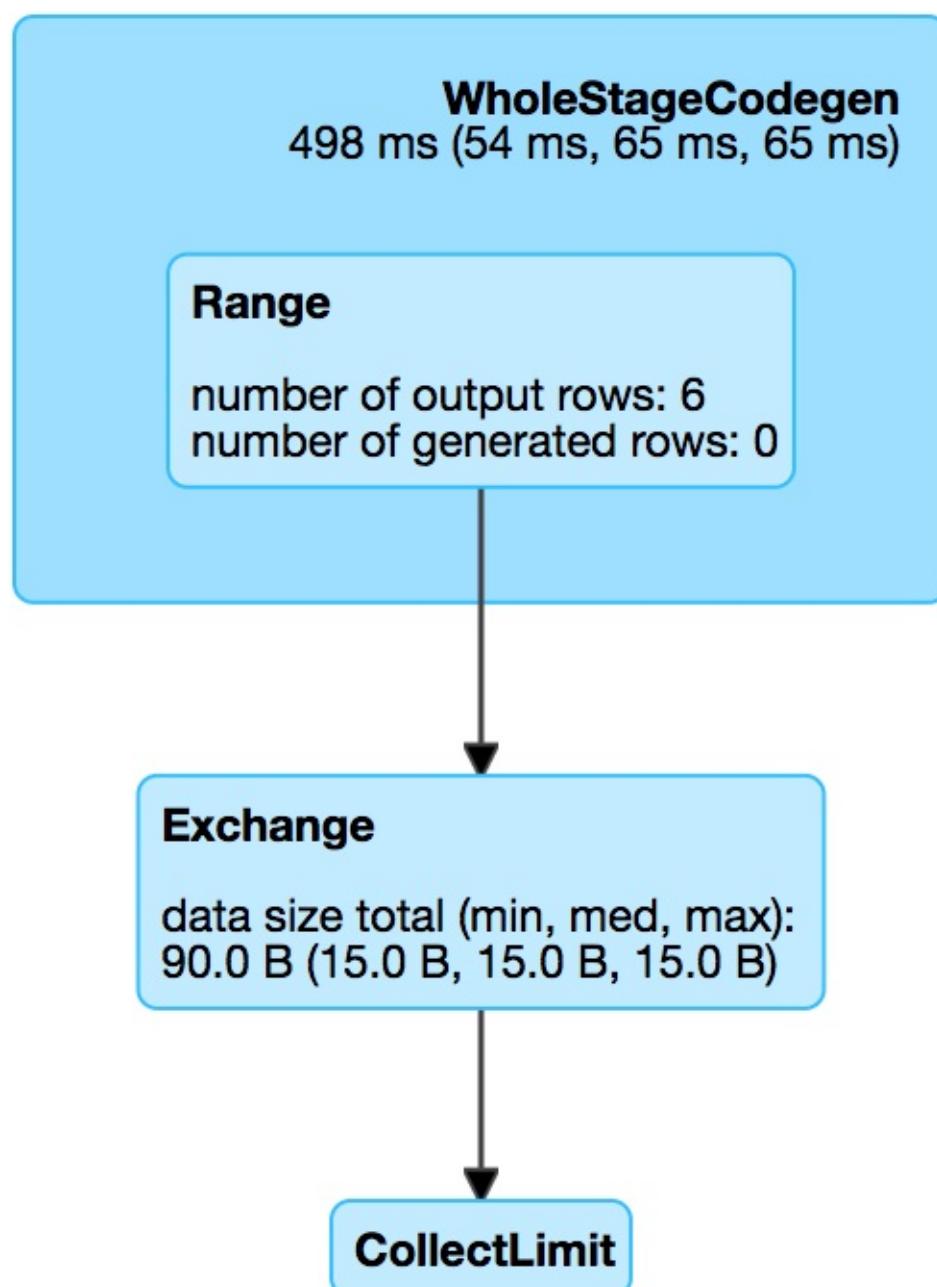


Figure 1. ShuffleExchangeExec in web UI (Details for Query)

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

```
doExecute(): RDD[InternalRow]
```

Note `doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` creates a new [ShuffledRowRDD](#) or (re)uses the [cached one](#) if `doExecute` was executed before.

Note `ShuffleExchangeExec` caches a [ShuffledRowRDD](#) for later reuse.

`doExecute` branches off per the optional [ExchangeCoordinator](#).

Note The optional [ExchangeCoordinator](#) is available only when [Adaptive Query Execution](#) is enabled (and `EnsureRequirements` physical query optimization is requested to [enforce partition requirements \(distribution and ordering\)](#) of a [physical operator](#)).

If `ExchangeCoordinator` was available, `doExecute` requests the [ExchangeCoordinator](#) for a [ShuffledRowRDD](#).

Otherwise (when no `ExchangeCoordinator` is available), `doExecute` [prepareShuffleDependency](#) and [preparePostShuffleRDD](#).

In the end, `doExecute` saves ([caches](#)) the result [ShuffledRowRDD](#) (as [cachedShuffleRDD](#) internal registry).

preparePostShuffleRDD Method

```
preparePostShuffleRDD(  
    shuffleDependency: ShuffleDependency[Int, InternalRow, InternalRow],  
    specifiedPartitionStartIndices: Option[Array[Int]] = None): ShuffledRowRDD
```

`preparePostShuffleRDD` ...FIXME

Note

`preparePostShuffleRDD` is used when:

- `ExchangeCoordinator` is requested to [doEstimationIfNecessary](#)
- `ShuffleExchangeExec` physical operator is requested to [execute](#)

prepareShuffleDependency Internal Method

```
prepareShuffleDependency(): ShuffleDependency[Int, InternalRow, InternalRow]
```

prepareShuffleDependency ...FIXME

Note

prepareShuffleDependency is used when:

- ExchangeCoordinator is requested to doEstimationIfNecessary (when ExchangeCoordinator is requested for a post-shuffle RDD ([ShuffledRowRDD](#)))
- ShuffleExchangeExec physical operator is requested to execute

prepareShuffleDependency Helper Method

```
prepareShuffleDependency(
    rdd: RDD[InternalRow],
    outputAttributes: Seq[Attribute],
    newPartitioning: Partitioning,
    serializer: Serializer): ShuffleDependency[Int, InternalRow, InternalRow]
```

prepareShuffleDependency creates a [ShuffleDependency](#) dependency.

Note

prepareShuffleDependency is used when `shuffleExchangeExec` prepares a `ShuffleDependency` (as part of...FIXME), `collectLimitExec` and `TakeOrderedAndProjectExec` physical operators are executed.

Preparing Physical Operator for Execution — doPrepare Method

```
doPrepare(): Unit
```

Note

`doPrepare` is part of [SparkPlan Contract](#) to prepare a physical operator for execution.

`doPrepare` simply requests the `ExchangeCoordinator` to register the `ShuffleExchangeExec` unary physical operator.

Internal Properties

Table 2. ShuffleExchangeExec's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
cachedShuffleRDD	ShuffledRowRDD that is created when <code>ShuffleExchangeExec</code> operator is executed (to generate RDD[InternalRow]) and reused (<i>cached</i>) if the operator is used by multiple plans
serializer	<code>UnsafeRowSerializer</code> (of the size as the number of the output schema attributes of the child physical operator and the dataSize performance metric) Used exclusively in <code>prepareShuffleDependency</code> to create a <code>ShuffleDependency</code>

ShuffledHashJoinExec Binary Physical Operator for Shuffled Hash Join

`ShuffledHashJoinExec` is a [binary physical operator](#) to execute a [shuffled hash join](#).

`ShuffledHashJoinExec` performs a hash join of two child relations by first shuffling the data using the join keys.

`ShuffledHashJoinExec` is [selected](#) to represent a [Join](#) logical operator when [JoinSelection](#) execution planning strategy is executed and `spark.sql.join.preferSortMergeJoin` configuration property is off.

Note

`spark.sql.join.preferSortMergeJoin` is an internal configuration property and is enabled by default.

That means that [JoinSelection](#) execution planning strategy (and so Spark Planner) prefers [sort merge join](#) over shuffled hash join.

In other words, you will *hardly* see shuffled hash joins in your structured queries unless you turn `spark.sql.join.preferSortMergeJoin` on.

Beside the `spark.sql.join.preferSortMergeJoin` configuration property one of the following requirements has to hold:

- (For a right build side, i.e. `BuildRight`) [canBuildRight](#), [canBuildLocalHashMap](#) for the right join side and finally the right join side is [at least three times smaller](#) than the left side
- (For a right build side, i.e. `BuildRight`) Left join keys are **not** [orderable](#), i.e. cannot be sorted
- (For a left build side, i.e. `BuildLeft`) [canBuildLeft](#), [canBuildLocalHashMap](#) for left join side and finally left join side is [at least three times smaller](#) than right

Tip

Enable `DEBUG` logging level for `org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys` logger to see the join condition and the left and right join keys.

```
// Use ShuffledHashJoinExec's selection requirements
// 1. Disable auto broadcasting
// JoinSelection (canBuildLocalHashMap specifically) requires that
// plan.stats.sizeInBytes < autoBroadcastJoinThreshold * numShufflePartitions
// That gives that autoBroadcastJoinThreshold has to be at least 1
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 1)
```

```

scala> println(spark.sessionState.conf.numShufflePartitions)
200

// 2. Disable preference on SortMergeJoin
spark.conf.set("spark.sql.join.preferSortMergeJoin", false)

val dataset = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "ShuffledHashJoinExec")
).toDF("id", "token")
// Self LEFT SEMI join
val q = dataset.join(dataset, Seq("id"), "leftsemi")

val sizeInBytes = q.queryExecution.optimizedPlan.stats.sizeInBytes
scala> println(sizeInBytes)
72

// 3. canBuildLeft is on for leftsemi

// the right join side is at least three times smaller than the left side
// Even though it's a self LEFT SEMI join there are two different join sides
// How is that possible?

// BINGO! ShuffledHashJoin is here!

// Enable DEBUG logging level
import org.apache.log4j.{Level, Logger}
val logger = "org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys"
Logger.getLogger(logger).setLevel(Level.DEBUG)

// ShuffledHashJoin with BuildRight
scala> q.explain
== Physical Plan ==
ShuffledHashJoin [id#37], [id#41], LeftSemi, BuildRight
:- Exchange hashpartitioning(id#37, 200)
: +- LocalTableScan [id#37, token#38]
+- Exchange hashpartitioning(id#41, 200)
  +- LocalTableScan [id#41]

scala> println(q.queryExecution.executedPlan.numberedTreeString)
00 ShuffledHashJoin [id#37], [id#41], LeftSemi, BuildRight
01 :- Exchange hashpartitioning(id#37, 200)
02 : +- LocalTableScan [id#37, token#38]
03 +- Exchange hashpartitioning(id#41, 200)
04   +- LocalTableScan [id#41]

```

Table 1. ShuffledHashJoinExec's Performance Metrics

Key	Name (in web UI)	Description
avgHashProbe	avg hash probe	
buildDataSize	data size of build side	
buildTime	time to build hash map	
numOutputRows	number of output rows	

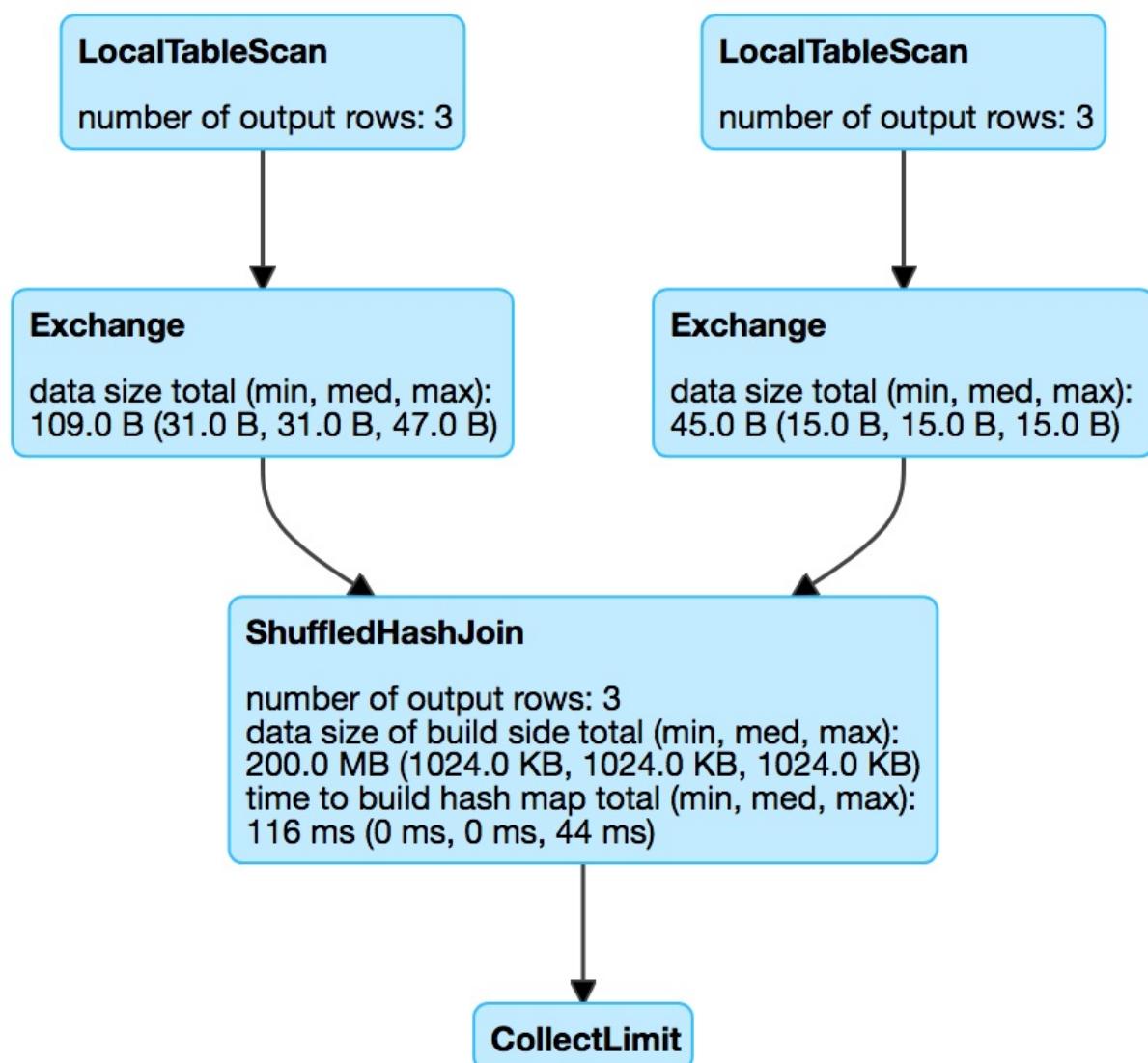


Figure 1. ShuffledHashJoinExec in web UI (Details for Query)

Table 2. ShuffledHashJoinExec's Required Child Output Distributions

Left Child	Right Child
HashClusteredDistribution (per left join key expressions)	HashClusteredDistribution (per right join key expressions)

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

```
doExecute(): RDD[InternalRow]
```

Note `doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` requests [streamedPlan](#) physical operator to [execute](#) (and generate a `RDD[InternalRow]`).

`doExecute` requests [buildPlan](#) physical operator to [execute](#) (and generate a `RDD[InternalRow]`).

`doExecute` requests [streamedPlan](#) physical operator's `RDD[InternalRow]` to zip partition-wise with [buildPlan](#) physical operator's `RDD[InternalRow]` (using `RDD.zipPartitions` method with `preservesPartitioning` flag disabled).

`doExecute` generates a `ZippedPartitionsRDD2` that you can see in a RDD lineage.

```
scala> println(q.queryExecution.toRdd.toDebugString)
(200) ZippedPartitionsRDD2[8] at toRdd at <console>:26 []
|   ShuffledRowRDD[3] at toRdd at <console>:26 []
+- (3) MapPartitionsRDD[2] at toRdd at <console>:26 []
|   MapPartitionsRDD[1] at toRdd at <console>:26 []
|   ParallelCollectionRDD[0] at toRdd at <console>:26 []
|   ShuffledRowRDD[7] at toRdd at <console>:26 []
+- (3) MapPartitionsRDD[6] at toRdd at <console>:26 []
|   MapPartitionsRDD[5] at toRdd at <console>:26 []
|   ParallelCollectionRDD[4] at toRdd at <console>:26 []
```

`doExecute` uses `RDD.zipPartitions` with a function applied to zipped partitions that takes two iterators of rows from the partitions of `streamedPlan` and `buildPlan`.

For every partition (and pairs of rows from the RDD), the function [buildHashedRelation](#) on the partition of `buildPlan` and [join](#) the `streamedPlan` partition iterator, the [HashedRelation](#), [numOutputRows](#) and [avgHashProbe](#) SQL metrics.

Building HashedRelation for Internal Rows — buildHashedRelation Internal Method

```
buildHashedRelation(iter: Iterator[InternalRow]): HashedRelation
```

`buildHashedRelation` creates a `HashedRelation` (for the input `iter` iterator of `InternalRows`, `buildKeys` and the current `TaskMemoryManager`).

Note

`buildHashedRelation` uses `TaskContext.get()` to access the current `TaskContext` that in turn is used to access the `TaskMemoryManager`.

`buildHashedRelation` records the time to create the `HashedRelation` as `buildTime`.

`buildHashedRelation` requests the `HashedRelation` for `estimatedSize` that is recorded as `buildDataSize`.

Note

`buildHashedRelation` is used exclusively when `ShuffledHashJoinExec` is requested to `execute` (when `streamedPlan` and `buildPlan` physical operators are executed and their RDDs zipped partition-wise using `RDD.zipPartitions` method).

Creating ShuffledHashJoinExec Instance

`ShuffledHashJoinExec` takes the following when created:

- Left join key `expressions`
- Right join key `expressions`
- `Join type`
- `BuildSide`
- Optional join condition `expression`
- Left `physical operator`
- Right `physical operator`

SerializeFromObjectExec Unary Physical Operator

`SerializeFromObjectExec` is a [unary physical operator](#) (i.e. with one [child](#) physical operator) that supports [Java code generation](#).

`SerializeFromObjectExec` supports Java code generation with the [doProduce](#), [doConsume](#) and [inputRDDs](#) methods.

`SerializeFromObjectExec` is a [ObjectConsumerExec](#).

`SerializeFromObjectExec` is [created](#) exclusively when [BasicOperators](#) execution planning strategy is requested to [plan](#) a `SerializeFromObject` logical operator.

`SerializeFromObjectExec` uses the [child](#) physical operator when requested for the [input RDDs](#) and the [outputPartitioning](#).

`SerializeFromObjectExec` uses the [serializer](#) for the [output schema attributes](#).

Creating `SerializeFromObjectExec` Instance

`SerializeFromObjectExec` takes the following when created:

- `Serializer` (as `Seq[NamedExpression]`)
- Child [physical operator](#) (that supports [Java code generation](#))

Generating Java Source Code for Consume Path in Whole-Stage Code Generation — `doConsume` Method

```
doConsume(ctx: CodegenContext, input: Seq[ExprCode], row: ExprCode): String
```

Note	<code>doConsume</code> is part of CodegenSupport Contract to generate the Java source code for consume path in Whole-Stage Code Generation.
------	---

`doConsume` ...FIXME

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

```
doProduce(ctx: CodegenContext): String
```

Note

`doProduce` is part of [CodegenSupport Contract](#) to generate the Java source code for [produce path](#) in Whole-Stage Code Generation.

`doProduce` ...FIXME

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` requests the `child` physical operator to `execute` (that triggers physical query planning and generates an `RDD[InternalRow]`) and transforms it by executing the following function on internal rows per partition with index (using `RDD.mapPartitionsWithIndexInternal` that creates another RDD):

1. Creates an [UnsafeProjection](#) for the `serializer`
2. Requests the `UnsafeProjection` to [initialize](#) (for the partition index)
3. Executes the `UnsafeProjection` on all internal binary rows in the partition

Note

`doExecute` (by `RDD.mapPartitionsWithIndexInternal`) adds a new `MapPartitionsRDD` to the `RDD` lineage. Use `RDD.toDebugString` to see the additional `MapPartitionsRDD`.

SortAggregateExec Aggregate Physical Operator for Sort-Based Aggregation

Caution	FIXME
---------	-------

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

```
doExecute(): RDD[InternalRow]
```

Note	doExecute is part of SparkPlan Contract to generate the runtime representation of a structured query as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	--

```
doExecute ...FIXME
```

SortMergeJoinExec Binary Physical Operator for Sort Merge Join

`SortMergeJoinExec` is a [binary physical operator](#) to [execute](#) a [sort merge join](#).

`ShuffledHashJoinExec` is [selected](#) to represent a [Join](#) logical operator when [JoinSelection](#) execution planning strategy is executed for joins with [left join keys](#) that are [orderable](#), i.e. that can be ordered (sorted).

	<p>A join key is orderable when is of one of the following data types:</p> <ul style="list-style-type: none"> • <code>NullType</code> • AtomicType (that represents all the available types except <code>NullType</code>, <code>StructType</code>, <code>ArrayType</code>, <code>UserDefinedType</code>, <code>MapType</code>, and <code>ObjectType</code>) • <code>StructType</code> with orderable fields • <code>ArrayType</code> of orderable type • <code>UserDefinedType</code> of orderable type
Note	<p>Therefore, a join key is not orderable when is of the following data type:</p> <ul style="list-style-type: none"> • <code>MapType</code> • <code>ObjectType</code>

	<p><code>spark.sql.join.preferSortMergeJoin</code> is an internal configuration property and is enabled by default.</p> <p>That means that JoinSelection execution planning strategy (and so Spark Planner) prefers sort merge join over shuffled hash join.</p>
--	--

`SortMergeJoinExec` supports [Java code generation](#) (aka `codegen`) for inner and cross joins.

Tip	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys</code> logger to see the join condition and the left and right join keys.</p>
-----	---

```

// Disable auto broadcasting so Broadcast Hash Join won't take precedence
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)

val tokens = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "SortMergeJoinExec")
).toDF("id", "token")

// all data types are orderable
scala> tokens.printSchema
root
|-- id: integer (nullable = false)
|-- token: string (nullable = true)

// Spark Planner prefers SortMergeJoin over Shuffled Hash Join
scala> println(spark.conf.get("spark.sql.join.preferSortMergeJoin"))
true

val q = tokens.join(tokens, Seq("id"), "inner")
scala> q.explain
== Physical Plan ==
*(3) Project [id#5, token#6, token#10]
+- *(3) SortMergeJoin [id#5], [id#9], Inner
   :- *(1) Sort [id#5 ASC NULLS FIRST], false, 0
     :  +- Exchange hashpartitioning(id#5, 200)
       :    +- LocalTableScan [id#5, token#6]
   +- *(2) Sort [id#9 ASC NULLS FIRST], false, 0
      +- ReusedExchange [id#9, token#10], Exchange hashpartitioning(id#5, 200)

```

Table 1. SortMergeJoinExec's Performance Metrics

Key	Name (in web UI)	Description
numOutputRows	number of output rows	

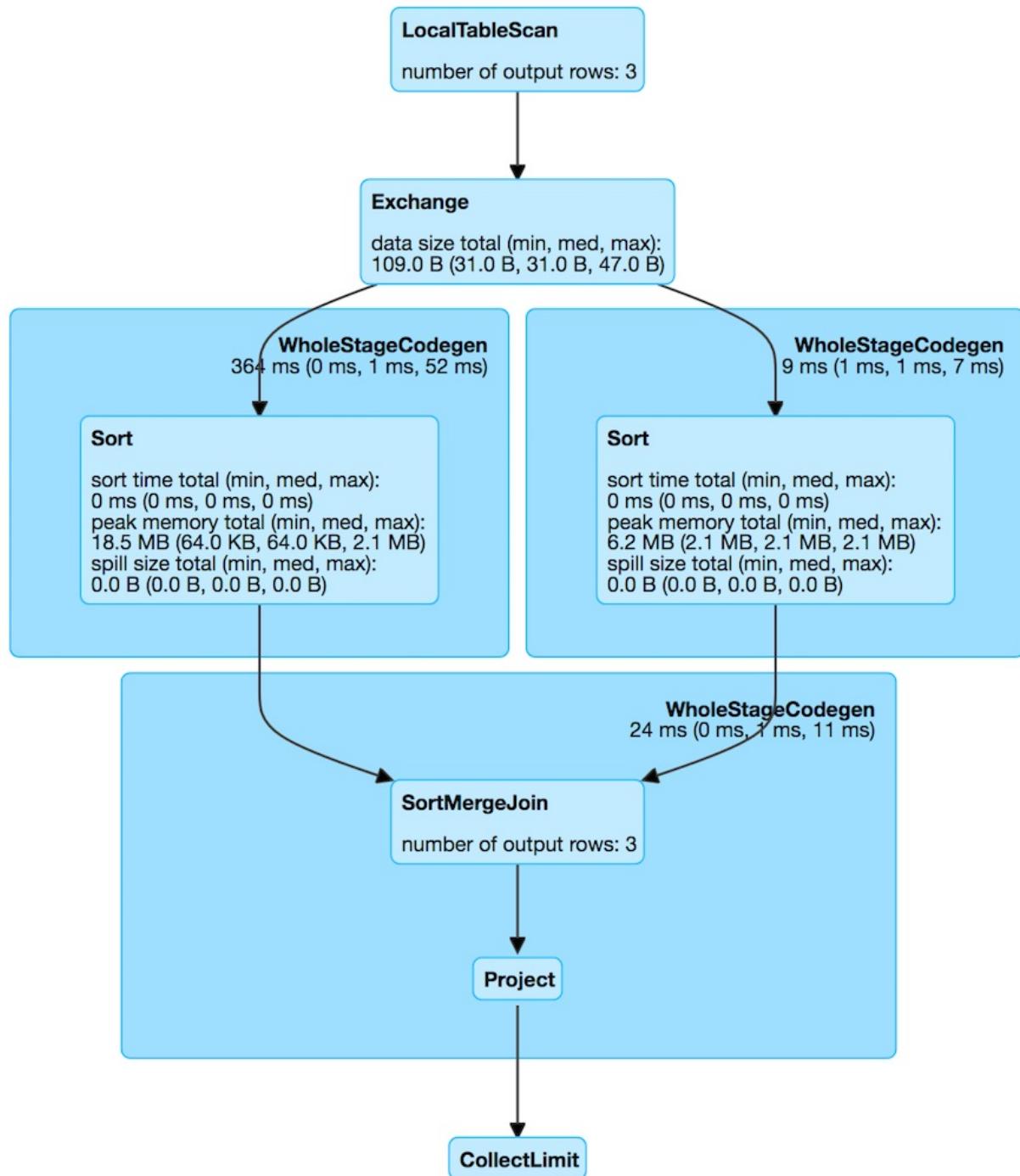


Figure 1. SortMergeJoinExec in web UI (Details for Query)

Note

The prefix for variable names for `SortMergeJoinExec` operators in `CodegenSupport`-generated code is **smj**.

```

scala> q.queryExecution.debug.codegen
Found 3 WholeStageCodegen subtrees.
== Subtree 1 / 3 ==
*Project [id#5, token#6, token#11]
+- *SortMergeJoin [id#5], [id#10], Inner
  :- *Sort [id#5 ASC NULLS FIRST], false, 0
    : +- Exchange hashpartitioning(id#5, 200)
      :   +- LocalTableScan [id#5, token#6]
  +- *Sort [id#10 ASC NULLS FIRST], false, 0
    +- ReusedExchange [id#10, token#11], Exchange hashpartitioning(id#5, 200)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */   return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {
/* 006 */   private Object[] references;
/* 007 */   private scala.collection.Iterator[] inputs;
/* 008 */   private scala.collection.Iterator smj_leftInput;
/* 009 */   private scala.collection.Iterator smj_rightInput;
/* 010 */   private InternalRow smj_leftRow;
/* 011 */   private InternalRow smj_rightRow;
/* 012 */   private int smj_value2;
/* 013 */   private org.apache.spark.sql.execution.ExternalAppendOnlyUnsafeRowArray smj_matches;
/* 014 */   private int smj_value3;
/* 015 */   private int smj_value4;
/* 016 */   private UTF8String smj_value5;
/* 017 */   private boolean smj_isNull2;
/* 018 */   private org.apache.spark.sql.execution.metric.SQLMetric smj_numOutputRows;
/* 019 */   private UnsafeRow smj_result;
/* 020 */   private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder smj_holder;
/* 021 */   private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter smj_rowWriter;
...

```

The output schema of a `SortMergeJoinExec` is...FIXME

The outputPartitioning of a `SortMergeJoinExec` is...FIXME

The outputOrdering of a `SortMergeJoinExec` is...FIXME

The partitioning requirements of the input of a `SortMergeJoinExec` (aka *child output distributions*) are HashClusteredDistributions of `left` and `right` join keys.

Table 2. SortMergeJoinExec's Required Child Output Distributions

Left Child	Right Child
HashClusteredDistribution (per <code>left join key expressions</code>)	HashClusteredDistribution (per <code>right join key expressions</code>)

The ordering requirements of the input of a `sortMergeJoinExec` (aka *child output ordering*) is...FIXME

Note	<code>SortMergeJoinExec</code> operator is chosen in <code>JoinSelection</code> execution planning strategy (after <code>BroadcastHashJoinExec</code> and <code>ShuffledHashJoinExec</code> physical join operators have not met the requirements).
------	---

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

```
doProduce(ctx: CodegenContext): String
```

Note	<code>doProduce</code> is part of <code>CodegenSupport Contract</code> to generate the Java source code for <code>produce path</code> in Whole-Stage Code Generation.
------	---

`doProduce` ...FIXME

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	<code>doExecute</code> is part of <code>SparkPlan Contract</code> to generate the runtime representation of a structured query as a distributed computation over <code>internal binary rows</code> on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	---

`doExecute` ...FIXME

Creating SortMergeJoinExec Instance

`SortMergeJoinExec` takes the following when created:

- Left join key `expressions`
- Right join key `expressions`
- Join type

- Optional join condition [expression](#)
- Left [physical operator](#)
- Right [physical operator](#)

SortExec Unary Physical Operator

SortExec is a [unary physical operator](#) that is created when:

- BasicOperators execution planning strategy is requested to [plan](#) a Sort logical operator
- FileFormatWriter helper object is requested to [write](#) the result of a structured query
- EnsureRequirements physical query optimization is executed (and [enforces](#) partition requirements for data distribution and ordering of a physical operator)

SortExec supports [Java code generation](#) (aka `codegen`).

```

val q = Seq((0, "zero"), (1, "one")).toDF("id", "name").sort('id)
val qe = q.queryExecution

val logicalPlan = qe.analyzed
scala> println(logicalPlan.numberedTreeString)
00 Sort [id#72 ASC NULLS FIRST], true
01 +- Project [_1#69 AS id#72, _2#70 AS name#73]
02   +- LocalRelation [_1#69, _2#70]

// BasicOperators does the conversion of Sort logical operator to SortExec
val sparkPlan = qe.sparkPlan
scala> println(sparkPlan.numberedTreeString)
00 Sort [id#72 ASC NULLS FIRST], true, 0
01 +- LocalTableScan [id#72, name#73]

// SortExec supports Whole-Stage Code Generation
val executedPlan = qe.executedPlan
scala> println(executedPlan.numberedTreeString)
00 *(1) Sort [id#72 ASC NULLS FIRST], true, 0
01 +- Exchange rangepartitioning(id#72 ASC NULLS FIRST, 200)
02   +- LocalTableScan [id#72, name#73]

import org.apache.spark.sql.execution.SortExec
val sortExec = executedPlan.collect { case se: SortExec => se }.head
assert(sortExec.isInstanceOf[SortExec])

```

When requested for the [output attributes](#), SortExec simply gives whatever the child operator uses.

SortExec uses the [sorting order expressions](#) for the output data ordering requirements.

When requested for the [output data partitioning requirements](#), SortExec simply gives whatever the child operator uses.

When requested for the [required partition requirements](#), `SortExec` gives the [OrderedDistribution](#) (with the sorting order expressions for the ordering) when the [global](#) flag is enabled (`true`) or the [UnspecifiedDistribution](#).

`SortExec` operator uses the [spark.sql.sort.enableRadixSort](#) internal configuration property (enabled by default) to control...FIXME

Table 1. SortExec's Performance Metrics

Key	Name (in web UI)	Description
<code>peakMemory</code>	peak memory	
<code>sortTime</code>	sort time	
<code>spillSize</code>	spill size	

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

```
doProduce(ctx: CodegenContext): String
```

Note

`doProduce` is part of [CodegenSupport Contract](#) to generate the Java source code for [produce path](#) in Whole-Stage Code Generation.

`doProduce` ...FIXME

Creating SortExec Instance

`SortExec` takes the following when created:

- [Sorting order expressions](#) (`Seq[SortOrder]`)
- `global` flag
- Child [physical plan](#)
- `testSpillFrequency` (default: `0`)

`createSorter` Method

```
createSorter(): UnsafeExternalRowSorter
```

`createSorter` ...FIXME

Note	createSorter is used when...FIXME
------	-----------------------------------

SubqueryExec Unary Physical Operator

`SubqueryExec` is a [unary physical operator](#) (i.e. with one [child](#) physical operator) that...

FIXME

`SubqueryExec` uses [relationFuture](#) that is lazily and executed only once when `SubqueryExec` is first requested to [prepare execution](#) that simply triggers execution of the [child](#) operator asynchronously (i.e. on a separate thread) and to [collect the result](#) soon after (that makes `SubqueryExec` waiting indefinitely for the child operator to be finished).

Caution

FIXME When is `doPrepare` executed?

`SubqueryExec` is [created](#) exclusively when `PlanSubqueries` preparation rule is [executed](#) (and transforms `ScalarSubquery` expressions in a physical plan).

```
val q = sql("select (select max(id) from t1) tt from t1")
scala> q.explain
== Physical Plan ==
*Project [Subquery subquery32 AS tt#33L]
: +- Subquery subquery32
:   +- *HashAggregate(keys=[], functions=[max(id#20L)])
:     +- Exchange SinglePartition
:       +- *HashAggregate(keys=[], functions=[partial_max(id#20L)])
:         +- *FileScan parquet default.t1[id#20L] Batched: true, Format: Parquet,
Location: InMemoryFileIndex[file:/Users/jacek/dev/oss/spark/spark-warehouse/t1], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:bigint>
+- *FileScan parquet default.t1[] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/Users/jacek/dev/oss/spark/spark-warehouse/t1], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<>
```

Table 1. SubqueryExec's Performance Metrics

Key	Name (in web UI)	Description
<code>collectTime</code>	time to collect (ms)	
<code>dataSize</code>	data size (bytes)	

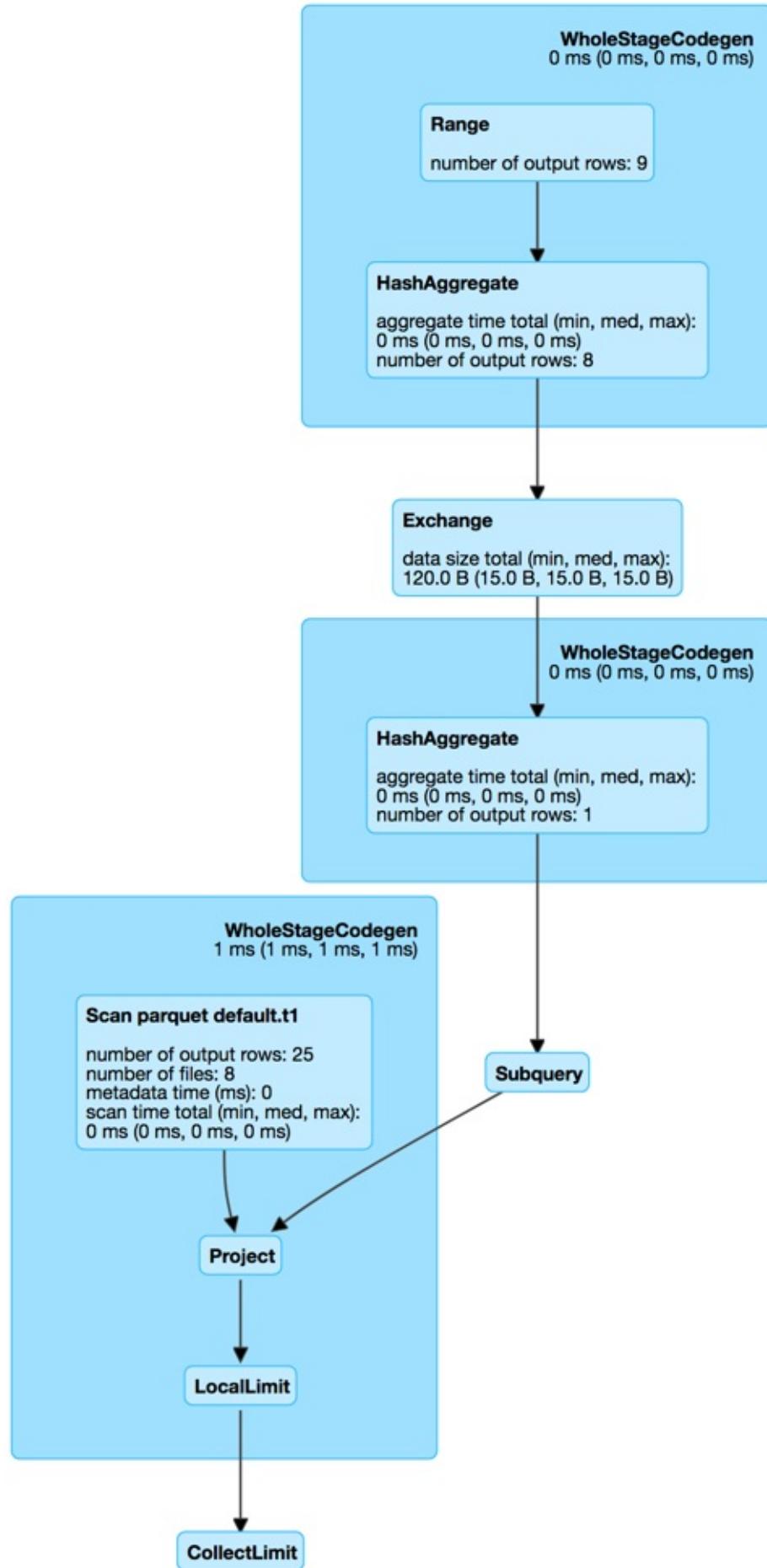


Figure 1. SubqueryExec in web UI (Details for Query)

Note

`SubqueryExec` physical operator is *almost* an exact copy of `BroadcastExchangeExec` physical operator.

Executing Child Operator Asynchronously — `doPrepare` Method

```
doPrepare(): Unit
```

Note

`doPrepare` is part of [SparkPlan Contract](#) to prepare a physical operator for execution.

`doPrepare` simply triggers initialization of the internal lazily-once-initialized `relationFuture` asynchronous computation.

`relationFuture` Internal Lazily-Once-Initialized Property

```
relationFuture: Future[Array[InternalRow]]
```

When "materialized" (aka `executed`), `relationFuture` spawns a new thread of execution that requests `SQLExecution` to execute an action (with the current `execution id`) on [subquery daemon cached thread pool](#).

Note

`relationFuture` uses Scala's [scala.concurrent.Future](#) that spawns a new thread of execution once instantiated.

The action tracks execution of the `child` physical operator to [executeCollect](#) and collects `collectTime` and `dataSize` SQL metrics.

In the end, `relationFuture` [posts metric updates](#) and returns the internal rows.

Note

`relationFuture` is executed on a separate thread from a custom [scala.concurrent.ExecutionContext](#) (built from a cached `java.util.concurrent.ThreadPoolExecutor` with the prefix `subquery` and up to 16 threads).

Note

`relationFuture` is used when `SubqueryExec` is requested to [prepare for execution](#) (that triggers execution of the child operator) and [execute collect](#) (that waits indefinitely until the child operator has finished).

Creating SubqueryExec Instance

`SubqueryExec` takes the following when created:

- Name of the subquery
- Child [physical plan](#)

Collecting Internal Rows of Executing SubqueryExec Operator — `executeCollect` Method

```
executeCollect(): Array[InternalRow]
```

Note	<code>executeCollect</code> is part of SparkPlan Contract to execute a physical operator and collect the results as collection of internal rows.
------	--

`executeCollect` waits till [relationFuture](#) gives a result (as a `Array[InternalRow]`).

InputAdapter Unary Physical Operator

`InputAdapter` is a [unary physical operator](#) (i.e. with one `child` physical operator) that is an adapter for the `child` physical operator that does not meet the requirements of [whole-stage Java code generation](#) (possibly due to `supportCodegen` flag turned off) but is between operators that participate in whole-stage Java code generation optimization.

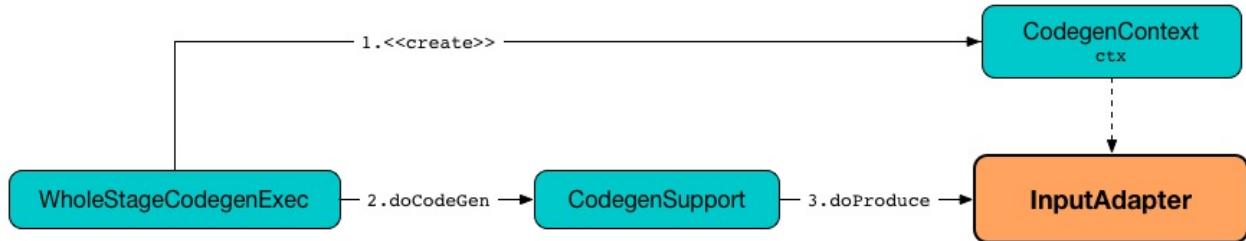


Figure 1. `InputAdapter`'s `doProduce`

`InputAdapter` takes a single `child` [physical plan](#) when created.

`InputAdapter` is [created](#) exclusively when `collapseCodegenStages` is requested to [insert](#) `InputAdapters` into a physical query plan with whole-stage Java code generation enabled.

`InputAdapter` makes sure that the prefix in the *text representation* of a physical plan tree is an empty string (and so it removes the star from the tree representation that `WholeStageCodegenExec` adds), e.g. for `explain` or `TreeNode.numberedTreeString` operators.

Tip

The number of `InputAdapters` is exactly the number of subtrees in a physical query plan that do not have stars.

```

scala> println(plan.numberedTreeString)
00 *(1) Project [id#117L]
01 +- *(1) BroadcastHashJoin [id#117L], [cast(id#115 as bigint)], Inner, BuildRight
02   :- *(1) Range (0, 1, step=1, splits=8)
03   +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false]
as bigint)))
04     +- Generate explode(ids#112), false, [id#115]
05       +- LocalTableScan [ids#112]
  
```

`InputAdapter` requires that...FIXME, i.e. `needCopyResult` flag is turned off.

`InputAdapter` [executes](#) the `child` physical operator to get the one and only one `RDD[InternalRow]` as its own input `RDDs` for [whole-stage produce path code generation](#).

```
// explode expression (that uses Generate operator) does not support codegen
val ids = Seq(Seq(0,1,2,3)).toDF("ids").select(explode($"ids") as "id")
val q = spark.range(1).join(ids, "id")
// Use executedPlan
// This is after the whole-stage Java code generation optimization is applied to a physical plan
val plan = q.queryExecution.executedPlan
scala> println(plan.numberedTreeString)
00 *(1) Project [id#117L]
01 +- *(1) BroadcastHashJoin [id#117L], [cast(id#115 as bigint)], Inner, BuildRight
02   :- *(1) Range (0, 1, step=1, splits=8)
03     +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
04       +- Generate explode(ids#112), false, [id#115]
05         +- LocalTableScan [ids#112]

// Find all InputAdapters in the physical query plan
import org.apache.spark.sql.execution.InputAdapter
scala> plan.collect { case a: InputAdapter => a }.zipWithIndex.map { case (op, idx) =>
  s"$idx $op" }.foreach(println)
0) BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
+- Generate explode(ids#112), false, [id#115]
  +- LocalTableScan [ids#112]
```

Generating Java Source Code for Produce Path in Whole-Stage Code Generation — `doProduce` Method

`doProduce(ctx: CodegenContext): String`

Note	<code>doProduce</code> is part of CodegenSupport Contract to generate the Java source code for produce path in Whole-Stage Code Generation.
------	---

`doProduce` generates a Java source code that consumes [internal row](#) of a single input [RDD](#) one at a time (in a `while` loop).

Note	<code>doProduce</code> supports one input RDD only (that the single child physical operator creates when executed).
------	--

Internally, `doProduce` generates two `input` and `row` "fresh" terms and registers `input` as a mutable state (in the generated class).

`doProduce` gives a plain Java source code that uses `input` and `row` terms as well as the code from [consume](#) code generator to iterate over the [internal binary rows](#) from the first [input RDD](#) only.

```

val q = spark.range(1)
  .select(explode(lit((0 to 1).toArray)) as "n") // <-- explode expression does not support codegen
  .join(spark.range(2))
  .where($"n" === $id")
scala> q.explain
== Physical Plan ==
*BroadcastHashJoin [cast(n#4 as bigint)], [id#7L], Inner, BuildRight
:- *Filter isnotnull(n#4)
: +- Generate explode([0,1]), false, false, [n#4]
:     +- *Project
:         +- *Range (0, 1, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
    +- *Range (0, 2, step=1, splits=8)

val plan = q.queryExecution.executedPlan
import org.apache.spark.sql.execution.InputAdapter
// there are two InputAdapters (for Generate and BroadcastExchange operators) so get is safe
val adapter = plan.collectFirst { case a: InputAdapter => a }.get

import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext

import org.apache.spark.sql.execution.CodegenSupport
val code = adapter.produce(ctx, plan.asInstanceOf[CodegenSupport])
scala> println(code)

/*inputadapter_c5*/

while (inputadapter_input2.hasNext() && !stopEarly()) {
  InternalRow inputadapter_row2 = (InternalRow) inputadapter_input2.next();
  /*wholestagecodegen_c1*/

  append(inputadapter_row2);
  if (shouldStop()) return;
}

```

```
import org.apache.spark.sql.catalyst.plans.logical.Range
val r = Range(start = 0, end = 1, step = 1, numSlices = 1)
import org.apache.spark.sql.execution.RangeExec
val re = RangeExec(r)

import org.apache.spark.sql.execution.InputAdapter
val ia = InputAdapter(re)

import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext

// You cannot call doProduce directly
// CodegenSupport.parent is not set up
// and so consume will throw NPE (that's used in doProduce)
// That's why you're supposed to call produce final method that does this
import org.apache.spark.sql.execution.CodegenSupport
ia.produce(ctx, parent = ia.asInstanceOf[CodegenSupport])

// produce however will lead to java.lang.UnsupportedOperationException
// which is due to doConsume throwing it by default
// and InputAdapter does not override it!
// That's why InputAdapter has to be under a WholeStageCodegenExec-enabled physical operator
//     which happens in CollapseCodegenStages.insertWholeStageCodegen
//     when a physical operator is CodegenSupport and meets codegen requirements
//     CollapseCodegenStages.supportCodegen
//     Most importantly it is CodegenSupport with supportCodegen flag on
//     The following physical operators turn supportCodegen flag off (and require Input Adapter wrapper)
//     1. GenerateExec
//     1. HashAggregateExec with a ImperativeAggregate aggregate function expression
//     1. SortMergeJoinExec with InnerLike joins, i.e. CROSS and INNER
//     1. InMemoryTableScanExec with output schema with primitive types only,
//         i.e. BooleanType, ByteType, ShortType, IntegerType, LongType, FloatType, DoubleType

FIXME Make the code working
```

WindowExec Unary Physical Operator

`WindowExec` is a [unary physical operator](#) (i.e. with one [child](#) physical operator) for [window aggregation execution](#) (i.e. represents [Window](#) unary logical operator at execution time).

`WindowExec` is [created](#) exclusively when [BasicOperators](#) execution planning strategy resolves a [Window](#) unary logical operator.

```
// arguably the most trivial example
// just a dataset of 3 rows per group
// to demo how partitions and frames work
// note the rows per groups are not consecutive (in the middle)
val metrics = Seq(
  (0, 0, 0), (1, 0, 1), (2, 5, 2), (3, 0, 3), (4, 0, 1), (5, 5, 3), (6, 5, 0)
).toDF("id", "device", "level")
scala> metrics.show
+---+-----+-----+
| id|device|level|
+---+-----+-----+
|  0|     0|    0|
|  1|     0|    1|
|  2|     5|    2| // <- this row for device 5 is among the rows of device 0
|  3|     0|    3| // <- as above but for device 0
|  4|     0|    1| // <- almost as above but there is a group of two rows for device
   0
|  5|     5|    3|
|  6|     5|    0|
+---+-----+-----+

// create windows of rows to use window aggregate function over every window
import org.apache.spark.sql.expressions.Window
val rangeWithTwoDevicesById = Window.
  partitionBy('device).
  orderBy('id).
  rangeBetween(start = -1, end = Window.currentRow) // <- demo rangeBetween first
val sumOverRange = metrics.withColumn("sum", sum('level) over rangeWithTwoDevicesById)

// Logical plan with Window unary logical operator
val optimizedPlan = sumOverRange.queryExecution.optimizedPlan
scala> println(optimizedPlan)
Window [sum(cast(level#9 as bigint)) windowspecdefinition(device#8, id#7 ASC NULLS FIRST, RANGE BETWEEN 1 PRECEDING AND CURRENT ROW) AS sum#15L], [device#8], [id#7 ASC NULLS FIRST]
+- LocalRelation [id#7, device#8, level#9]

// Physical plan with WindowExec unary physical operator (shown as Window)
scala> sumOverRange.explain
== Physical Plan ==
Window [sum(cast(level#9 as bigint)) windowspecdefinition(device#8, id#7 ASC NULLS FIRST]
```

```

ST, RANGE BETWEEN 1 PRECEDING AND CURRENT ROW) AS sum#15L], [device#8], [id#7 ASC NULLS
FIRST]
+- *Sort [device#8 ASC NULLS FIRST, id#7 ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(device#8, 200)
    +- LocalTableScan [id#7, device#8, level#9]

// Going fairly low-level...you've been warned

val plan = sumOverRange.queryExecution.executedPlan
import org.apache.spark.sql.execution.window.WindowExec
val we = plan.asInstanceOf[WindowExec]
val windowRDD = we.execute()
scala> :type windowRDD
org.apache.spark.rdd.RDD[org.apache.spark.sql.catalyst.InternalRow]

scala> windowRDD.toDebugString
res0: String =
(200) MapPartitionsRDD[5] at execute at <console>:35 []
|  MapPartitionsRDD[4] at execute at <console>:35 []
|  ShuffledRowRDD[3] at execute at <console>:35 []
+- (7) MapPartitionsRDD[2] at execute at <console>:35 []
  |  MapPartitionsRDD[1] at execute at <console>:35 []
  |  ParallelCollectionRDD[0] at execute at <console>:35 []

// no computation on the source dataset has really occurred
// Let's trigger a RDD action
scala> windowRDD.first
res0: org.apache.spark.sql.catalyst.InternalRow = [0,2,5,2,2]

scala> windowRDD.foreach(println)
[0,2,5,2,2]
[0,0,0,0,0]
[0,5,5,3,3]
[0,6,5,0,3]
[0,1,0,1,1]
[0,3,0,3,3]
[0,4,0,1,4]

scala> sumOverRange.show
+---+-----+-----+---+
| id|device|level|sum|
+---+-----+-----+---+
|  2|      5|     2|   2|
|  5|      5|     3|   3|
|  6|      5|     0|   3|
|  0|      0|     0|   0|
|  1|      0|     1|   1|
|  3|      0|     3|   3|
|  4|      0|     1|   4|
+---+-----+-----+---+

// use rowsBetween
val rowsWithTwoDevicesById = Window.

```

```

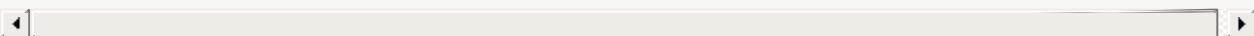
partitionBy('device).
orderBy('id).
rowsBetween(start = -1, end = Window.currentRow)
val sumOverRows = metrics.withColumn("sum", sum('level) over rowsWithTwoDevicesById)

// let's see the result first to have them close
// and compare row- vs range-based windows
scala> sumOverRows.show
+---+-----+-----+
| id|device|level|sum|
+---+-----+-----+
| 2|      5|     2|  2|
| 5|      5|     3|  5| <- a difference
| 6|      5|     0|  3|
| 0|      0|     0|  0|
| 1|      0|     1|  1|
| 3|      0|     3|  4| <- another difference
| 4|      0|     1|  4|
+---+-----+-----+

val rowsOptimizedPlan = sumOverRows.queryExecution.optimizedPlan
scala> println(rowsOptimizedPlan)
Window [sum(cast(level#901 as bigint)) windowspecdefinition(device#900, id#899 ASC NUL
LS FIRST, ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS sum#1458L], [device#900], [id#8
99 ASC NULLS FIRST]
+- LocalRelation [id#899, device#900, level#901]

scala> sumOverRows.explain
== Physical Plan ==
Window [sum(cast(level#901 as bigint)) windowspecdefinition(device#900, id#899 ASC NUL
LS FIRST, ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS sum#1458L], [device#900], [id#8
99 ASC NULLS FIRST]
+- *Sort [device#900 ASC NULLS FIRST, id#899 ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(device#900, 200)
    +- LocalTableScan [id#899, device#900, level#901]

```



```
// a more involved example
val dataset = spark.range(start = 0, end = 13, step = 1, numPartitions = 4)

import org.apache.spark.sql.expressions.Window
val groupsOrderById = Window.partitionBy('group).rangeBetween(-2, Window.currentRow).o
rderBy('id)
val query = dataset.
  withColumn("group", 'id % 4).
  select('*', sum('id) over groupsOrderById as "sum")

scala> query.explain
== Physical Plan ==
Window [sum(id#25L) windowspecdefinition(group#244L, id#25L ASC NULLS FIRST, RANGE BET
WEEN 2 PRECEDING AND CURRENT ROW) AS sum#249L], [group#244L], [id#25L ASC NULLS FIRST]
+- *Sort [group#244L ASC NULLS FIRST, id#25L ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(group#244L, 200)
    +- *Project [id#25L, (id#25L % 4) AS group#244L]
      +- *Range (0, 13, step=1, splits=4)

val plan = query.queryExecution.executedPlan
import org.apache.spark.sql.execution.window.WindowExec
val we = plan.asInstanceOf[WindowExec]
```

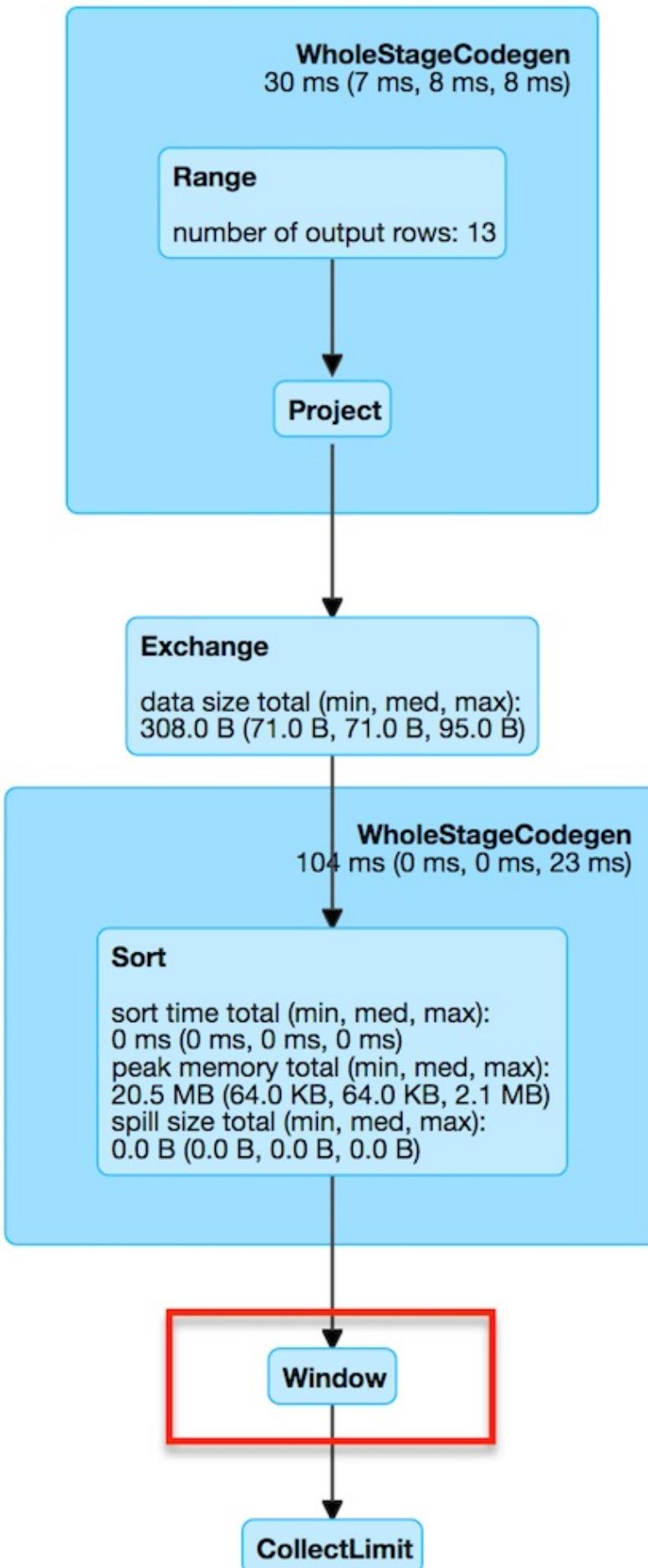


Figure 1. WindowExec in web UI (Details for Query)

The output schema of `WindowExec` are the attributes of the child physical operator and the window expressions.

```
// the whole schema is as follows
val schema = query.queryExecution.executedPlan.output.toStructType
scala> println(schema.treeString)
root
|-- id: long (nullable = false)
|-- group: long (nullable = true)
|-- sum: long (nullable = true)

// Let's see ourselves how the schema is made up of

scala> :type we
org.apache.spark.sql.execution.window.WindowExec

// child's output
scala> println(we.child.output.toStructType.treeString)
root
|-- id: long (nullable = false)
|-- group: long (nullable = true)

// window expressions' output
val weExprSchema = we.windowExpression.map(_.toAttribute).toStructType
scala> println(weExprSchema.treeString)
root
|-- sum: long (nullable = true)
```

The required child output distribution of a `WindowExec` operator is one of the following:

- AllTuples when the window partition specification expressions is empty
- ClusteredDistribution (with the window partition specification expressions) with the partition specification specified

If no window partition specification is specified, `WindowExec` prints out the following WARN message to the logs:

```
WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
```

Enable `WARN` logging level for `org.apache.spark.sql.execution.WindowExec` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

Tip

```
log4j.logger.org.apache.spark.sql.execution.WindowExec=WARN
```

Refer to [Logging](#).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` [executes](#) the single `child` physical operator and [maps over partitions](#) using a custom `Iterator[InternalRow]` .

Note

When executed, `doExecute` creates a `MapPartitionsRDD` with the `child` physical operator's `RDD[InternalRow]` .

```
scala> :type we
org.apache.spark.sql.execution.window.WindowExec

val windowRDD = we.execute
scala> :type windowRDD
org.apache.spark.rdd.RDD[org.apache.spark.sql.catalyst.InternalRow]

scala> println(windowRDD.toDebugString)
(200) MapPartitionsRDD[5] at execute at <console>:35 []
|  MapPartitionsRDD[4] at execute at <console>:35 []
|  ShuffledRowRDD[3] at execute at <console>:35 []
+- (7) MapPartitionsRDD[2] at execute at <console>:35 []
   |  MapPartitionsRDD[1] at execute at <console>:35 []
   |  ParallelCollectionRDD[0] at execute at <console>:35 []
```

Internally, `doExecute` first takes [WindowExpressions](#) and their [WindowFunctionFrame](#) factory functions (from [window frame factories](#)) followed by [executing](#) the single `child` physical operator and mapping over partitions (using `RDD.mapPartitions` operator).

`doExecute` creates an `Iterator[InternalRow]` (of [UnsafeRow](#) exactly).

Mapping Over UnsafeRows per Partition

— Iterator[InternalRow]

When created, `Iterator[InternalRow]` first creates two [UnsafeProjection](#) conversion functions (to convert `InternalRow`s to `UnsafeRow`s) as `result` and `grouping`.

Note	<code>grouping</code> conversion function is created for window partition specifications expressions and used exclusively to create <code>nextGroup</code> when <code>Iterator[InternalRow]</code> is requested <code>nextRow</code> .
------	--

Tip	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator</code> logger to see the code generated for <code>grouping</code> conversion function.</p>
-----	---

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator=DEB
```

Refer to [Logging](#).

`Iterator[InternalRow]` then fetches the first row from the upstream RDD and initializes `nextRow` and `nextGroup` [UnsafeRow](#)s.

Note	<code>nextGroup</code> is the result of converting <code>nextRow</code> using <code>grouping</code> conversion function.
------	--

`doExecute` creates a [ExternalAppendOnlyUnsafeRowArray](#) buffer using `spark.sql.windowExec.buffer.spill.threshold` property (default: `4096`) as the threshold for the number of rows buffered.

`doExecute` creates a `SpecificInternalRow` for the window function result (as `windowFunctionResult`).

Note	<code>SpecificInternalRow</code> is also used in the generated code for the <code>UnsafeProjection</code> for the result.
------	---

`doExecute` takes the window frame factories and generates [WindowFunctionFrame](#) per factory (using the `SpecificInternalRow` created earlier).

Caution	FIXME
---------	-------

Note	<code>ExternalAppendOnlyUnsafeRowArray</code> is used to collect <code>UnsafeRow</code> objects from the child's partitions (one partition per buffer and up to <code>spark.sql.windowExec.buffer.spill.threshold</code>).
------	---

next Method

```
override final def next(): InternalRow
```

Note	<code>next</code> is part of Scala's scala.collection.Iterator interface that returns the next element and discards it from the iterator.
------	---

`next` method of the final `Iterator` is...FIXME

`next` first [fetches a new partition](#), but only when...FIXME

Note	<code>next</code> loads all the rows in <code>nextGroup</code> .
------	--

Caution	FIXME What's <code>nextGroup</code> ?
---------	---------------------------------------

`next` takes one [UnsafeRow](#) from `bufferIterator`.

Caution	FIXME <code>bufferIterator</code> seems important for the iteration.
---------	--

`next` then requests every [WindowFunctionFrame](#) to write the current `rowIndex` and `UnsafeRow`.

Caution	FIXME <code>rowIndex</code> ?
---------	-------------------------------

`next` joins the current `UnsafeRow` and `windowFunctionResult` (i.e. takes two `InternalRows` and makes them appear as a single concatenated `InternalRow`).

`next` increments `rowIndex`.

In the end, `next` uses the `unsafeProjection` function (that was created using [createResultProjection](#)) and projects the joined `InternalRow` to the result `UnsafeRow`.

Fetching All Rows In Partition — `fetchNextPartition` Internal Method

```
fetchNextPartition(): Unit
```

`fetchNextPartition` first copies the current `nextGroup UnsafeRow` (that was created using [grouping](#) projection function) and clears the internal `buffer`.

`fetchNextPartition` then collects all `UnsafeRows` for the current `nextGroup` in `buffer`.

With the `buffer` filled in (with `UnsafeRows` per partition), `fetchNextPartition` prepares every [WindowFunctionFrame](#) function in `frames` one by one (and passing `buffer`).

In the end, `fetchNextPartition` resets `rowIndex` to `0` and requests `buffer` to generate an iterator (available as `bufferIterator`).

Note

`fetchNextPartition` is used internally when `doExecute`'s `Iterator` is requested for the `next UnsafeRow` (when `bufferIterator` is uninitialized or was drained, i.e. holds no elements, but there are still rows in the upstream operator's partition).

fetchNextRow Internal Method

```
fetchNextRow(): Unit
```

`fetchNextRow` checks whether there is the next row available (using the upstream `Iterator.hasNext`) and sets `nextRowAvailable` mutable internal flag.

If there is a row available, `fetchNextRow` sets `nextRow` internal variable to the `next UnsafeRow` from the upstream's RDD.

`fetchNextRow` also sets `nextGroup` internal variable as an `UnsafeRow` for `nextRow` using `grouping` function.

Note

`grouping` is a `UnsafeProjection` function that is `created` for window partition specifications expressions to be bound to the single `child`'s output schema.

`grouping` uses `GenerateUnsafeProjection` to canonicalize the bound expressions and `create` the `UnsafeProjection` function.

If no row is available, `fetchNextRow` nullifies `nextRow` and `nextGroup` internal variables.

Note

`fetchNextRow` is used internally when `doExecute`'s `Iterator` is created and `fetchNextPartition` is called.

createResultProjection Internal Method

```
createResultProjection(expressions: Seq[Expression]): UnsafeProjection
```

`createResultProjection` creates a `UnsafeProjection` function for `expressions` window function `Catalyst expressions` so that the window expressions are on the right side of child's output.

Note

`UnsafeProjection` is a Scala function that produces `UnsafeRow` for an `InternalRow`.

Internally, `createResultProjection` first creates a translation table with a `BoundReference` per expression (in the input `expressions`).

Note

`BoundReference` is a Catalyst expression that is a reference to a value in `internal binary row` at a specified position and of specified data type.

`createResultProjection` then creates a window function bound references for `window expressions` so unbound expressions are transformed to the `BoundReferences`.

In the end, `createResultProjection` creates a `UnsafeProjection` with:

- `exprs` expressions from `child`'s output and the collection of window function bound references
- `inputSchema` input schema per `child`'s output

Note

`createResultProjection` is used exclusively when `WindowExec` is `executed`.

Creating WindowExec Instance

`WindowExec` takes the following when created:

- Window `named expressions`
- Window partition specification `expressions`
- Window order specification (as a collection of `SortOrder` expressions)
- Child `physical operator`

Lookup Table for WindowExpressions and Factory Functions for WindowFunctionFrame

— `windowFrameExpressionFactoryPairs` Lazy Value

```
windowFrameExpressionFactoryPairs:  
Seq[(mutable.Buffer[WindowExpression], InternalRow => WindowFunctionFrame)]
```

`windowFrameExpressionFactoryPairs` is a lookup table with `window expressions` and `factory functions` for `WindowFunctionFrame` (per key-value pair in `framedFunctions` lookup table).

A factory function is a function that takes an `InternalRow` and produces a `WindowFunctionFrame` (described in the table below)

Internally, `windowFrameExpressionFactoryPairs` first builds `framedFunctions` lookup table with 4-element tuple keys and 2-element expression list values (described in the table below).

`windowFrameExpressionFactoryPairs` finds [WindowExpression](#) expressions in the input `windowExpression` and for every `WindowExpression` takes the `window frame specification` (of type `SpecifiedWindowFrame`) that is used to find frame type and start and end frame positions).

Table 1. `framedFunctions`'s FrameKey—4-element Tuple for Frame Keys (in positional order)

Element	Description
Name of the kind of function	<ul style="list-style-type: none"> • AGGREGATE for AggregateFunction (in AggregateExpressions) or AggregateWindowFunction • OFFSET for <code>OffsetWindowFunction</code>
FrameType	<code>RangeFrame</code> OR <code>RowFrame</code>
Window frame's start position	<ul style="list-style-type: none"> • Positive number for <code>currentRow</code> (0) and <code>ValueFollowing</code> • Negative number for <code>valuePreceding</code> • Empty when unspecified
Window frame's end position	<ul style="list-style-type: none"> • Positive number for <code>currentRow</code> (0) and <code>ValueFollowing</code> • Negative number for <code>valuePreceding</code> • Empty when unspecified

Table 2. `framedFunctions`'s 2-element Tuple Values (in positional order)

Element	Description
Collection of window expressions	WindowExpression
Collection of window functions	<ul style="list-style-type: none"> • AggregateFunction (in AggregateExpressions) or AggregateWindowFunction • <code>OffsetWindowFunction</code>

`windowFrameExpressionFactoryPairs` creates a [AggregateProcessor](#) for `AGGREGATE` frame keys in `framedFunctions` lookup table.

Table 3. windowFrameExpressionFactoryPairs' Factory Functions (in creation order)

Frame Name	FrameKey	WindowFunctionFrame
Offset Frame	("OFFSET", RowFrame, Some(offset), Some(h))	OffsetWindowFunctionFrame
Growing Frame	("AGGREGATE", frameType, None, Some(high))	UnboundedPrecedingWindowFunctionFrame
Shrinking Frame	("AGGREGATE", frameType, Some(low), None)	UnboundedFollowingWindowFunctionFrame
Moving Frame	("AGGREGATE", frameType, Some(low), Some(high))	SlidingWindowFunctionFrame
Entire Partition Frame	("AGGREGATE", frameType, None, None)	UnboundedWindowFunctionFrame

Note	<code>lazy val</code> in Scala is computed when first accessed and once only (for the entire lifetime of the owning object instance).
Note	<code>windowFrameExpressionFactoryPairs</code> is used exclusively when <code>WindowExec</code> is executed .

createBoundOrdering Internal Method

```
createBoundOrdering(frame: FrameType, bound: Expression, timeZone: String): BoundOrdering
```

`createBoundOrdering` ...FIXME

Note	<code>createBoundOrdering</code> is used exclusively when <code>WindowExec</code> physical operator is requested for the window frame factories .
------	---

AggregateProcessor

`AggregateProcessor` is [created](#) and used exclusively when `WindowExec` physical operator is executed.

`AggregateProcessor` supports [DeclarativeAggregate](#) and [ImperativeAggregate](#) aggregate functions only (which happen to be [AggregateFunction](#) in [AggregateExpression](#) or [AggregateWindowFunction](#)).

Table 1. AggregateProcessor's Properties

Name	Description
<code>buffer</code>	<code>SpecificInternalRow</code> with data types given bufferSchema
Note	<code>AggregateProcessor</code> is created using <code>AggregateProcessor</code> factory object (using apply method).

initialize Method

```
initialize(size: Int): Unit
```

Caution	FIXME
Note	<p><code>initialize</code> is used when:</p> <ul style="list-style-type: none"> • <code>SlidingWindowFunctionFrame</code> writes out to the target row • <code>UnboundedWindowFunctionFrame</code> is prepared • <code>UnboundedPrecedingWindowFunctionFrame</code> is prepared • <code>UnboundedFollowingWindowFunctionFrame</code> writes out to the target row

evaluate Method

```
evaluate(target: InternalRow): Unit
```

Caution	FIXME
Note	<code>evaluate</code> is used when...FIXME

apply Factory Method

```
apply(
    functions: Array[Expression],
    ordinal: Int,
    inputAttributes: Seq[Attribute],
    newMutableProjection: (Seq[Expression], Seq[Attribute]) => MutableProjection): AggregateProcessor
```

Note

`apply` is used exclusively when `windowExec` is [executed](#) (and creates `WindowFunctionFrame` per `AGGREGATE` window aggregate functions, i.e. `AggregateExpression` or `AggregateWindowFunction`)

Executing update on ImperativeAggregates — update Method

```
update(input: InternalRow): Unit
```

`update` executes the [update](#) method on every input `ImperativeAggregate` sequentially (one by one).

Internally, `update` joins `buffer` with `input` [internal binary row](#) and converts the joined `InternalRow` using the [MutableProjection](#) function.

`update` then requests every `ImperativeAggregate` to [update](#) passing in the `buffer` and the input `input` rows.

Note

`MutableProjection` mutates the same underlying binary row object each time it is executed.

Note

`update` is used when `windowFunctionFrame` [prepares](#) or [writes](#).

Creating AggregateProcessor Instance

`AggregateProcessor` takes the following when created:

- Schema of the buffer (as a collection of `AttributeReferences`)
- Initial `MutableProjection`
- Update `MutableProjection`
- Evaluate `MutableProjection`

- [ImperativeAggregate](#) expressions for aggregate functions
- Flag whether to track partition size

WindowFunctionFrame

`WindowFunctionFrame` is a [contract](#) for...FIXME

Table 1. WindowFunctionFrame's Implementations

Name	Description
<code>OffsetWindowFunctionFrame</code>	
<code>SlidingWindowFunctionFrame</code>	
<code>UnboundedFollowingWindowFunctionFrame</code>	
<code>UnboundedPrecedingWindowFunctionFrame</code>	
<code>UnboundedWindowFunctionFrame</code>	

UnboundedWindowFunctionFrame

`UnboundedWindowFunctionFrame` is a [WindowFunctionFrame](#) that gives the same value for every row in a partition.

`UnboundedWindowFunctionFrame` is [created](#) for [AggregateFunctions](#) (in [AggregateExpressions](#)) or [AggregateWindowFunctions](#) with no frame defined (i.e. no `rowsBetween` or `rangeBetween`) that boils down to using the [entire partition frame](#).

`UnboundedWindowFunctionFrame` takes the following when created:

- Target [InternalRow](#)
- [AggregateProcessor](#)

prepare Method

```
prepare(rows: ExternalAppendOnlyUnsafeRowArray): Unit
```

`prepare` requests [AggregateProcessor](#) to [initialize](#) passing in the number of `UnsafeRows` in the input `ExternalAppendOnlyUnsafeRowArray`.

`prepare` then requests `ExternalAppendOnlyUnsafeRowArray` to generate an iterator.

In the end, `prepare` requests [AggregateProcessor](#) to [update](#) passing in every `UnsafeRow` in the iterator one at a time.

write Method

```
write(index: Int, current: InternalRow): Unit
```

`write` simply requests [AggregateProcessor](#) to [evaluate](#) the target [InternalRow](#).

WindowFunctionFrame Contract

```
package org.apache.spark.sql.execution.window

abstract class WindowFunctionFrame {
  def prepare(rows: ExternalAppendOnlyUnsafeRowArray): Unit
  def write(index: Int, current: InternalRow): Unit
}
```

Note

`WindowFunctionFrame` is a `private[window]` contract.

Table 2. WindowFunctionFrame Contract

Method	Description
<code>prepare</code>	Used exclusively when <code>WindowExec</code> operator fetches all UnsafeRows for a partition (passing in <code>ExternalAppendOnlyUnsafeRowArray</code> with all <code>UnsafeRows</code>).
<code>write</code>	Used exclusively when the <code>Iterator[InternalRow]</code> (from executing <code>WindowExec</code>) is requested a next row.

WholeStageCodegenExec Unary Physical Operator for Java Code Generation

`WholeStageCodegenExec` is a [unary physical operator](#) that is one of the two physical operators that lay the foundation for the [Whole-Stage Java Code Generation](#) for a [Codegened Execution Pipeline](#) of a structured query.

Note

[InputAdapter](#) is the other physical operator for Codegened Execution Pipeline of a structured query.

`WholeStageCodegenExec` itself supports the [Java code generation](#) and so when executed triggers code generation for the entire child physical plan subtree of a structured query.

```
val q = spark.range(10).where('id === 4)
scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*(1) Filter (id#3L = 4)
+- *(1) Range (0, 10, step=1, splits=8)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIteratorForCodegenStage1(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIteratorForCodegenStage1 extends org.apache.spark.sql.e
xecution.BufferedRowIterator {
...
}
```

Tip

Consider using [Debugging Query Execution facility](#) to deep dive into the whole-stage code generation.

```

val q = spark.range(10).where('id === 4)
import org.apache.spark.sql.execution.debug._
scala> q.debugCodegen()
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*(1) Filter (id#0L = 4)
+- *(1) Range (0, 10, step=1, splits=8)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIteratorForCodegenStage1(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIteratorForCodegenStage1 extends org.apache.spark.e
xecution.BufferedRowIterator {
...

```

Use the following to enable comments in generated code.

Tip

```
org.apache.spark.SparkEnv.get.conf.set("spark.sql.codegen.comments", "true")
```

```

val q = spark.range(10).where('id === 4)
import org.apache.spark.sql.execution.debug._
scala> q.debugCodegen()
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*(1) Filter (id#6L = 4)
+- *(1) Range (0, 10, step=1, splits=8)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIteratorForCodegenStage1(references);
/* 003 */ }
/* 004 */
/* 005 */ /**
 * Codegend pipeline for stage (id=1)
 * *(1) Filter (id#6L = 4)
 * +- *(1) Range (0, 10, step=1, splits=8)
 */
/* 006 */ final class GeneratedIteratorForCodegenStage1 extends org.apache.spark.e
xecution.BufferedRowIterator {
...

```

`WholeStageCodegenExec` is [created](#) when:

- `CollapseCodegenStages` physical query optimization is [executed](#) (with `spark.sql.codegen.wholeStage` configuration property enabled)

- `FileSourceScanExec` leaf physical operator is [executed](#) (with the `supportsBatch` flag enabled)
- `InMemoryTableScanExec` leaf physical operator is [executed](#) (with the `supportsBatch` flag enabled)
- `DataSourceV2ScanExec` leaf physical operator is [executed](#) (with the `supportsBatch` flag enabled)

Note `spark.sql.codegen.wholeStage` property is enabled by default.

`WholeStageCodegenExec` takes a single `child` physical operator (a physical subquery tree) and **codegen stage ID** when created.

Note `WholeStageCodegenExec` requires that the single `child` physical operator [supports Java code generation](#).

```
// RangeExec physical operator does support codegen
import org.apache.spark.sql.execution.RangeExec
import org.apache.spark.sql.catalyst.plans.logical.Range
val rangeExec = RangeExec(start = 0, end = 1, step = 1, numSlices = 1)

import org.apache.spark.sql.execution.WholeStageCodegenExec
val rdd = WholeStageCodegenExec(rangeExec)(codegenStageId = 0).execute()
```

`WholeStageCodegenExec` marks the `child` physical operator with `*` (star) prefix and [per-query codegen stage ID](#) (in round brackets) in the [text representation of a physical plan tree](#).

```
scala> println(plan.numberedTreeString)
00 *(1) Project [id#117L]
01 +- *(1) BroadcastHashJoin [id#117L], [cast(id#115 as bigint)], Inner, BuildRight
02   :- *(1) Range (0, 1, step=1, splits=8)
03   +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
04     +- Generate explode(ids#112), false, [id#115]
05       +- LocalTableScan [ids#112]
```

Note As `WholeStageCodegenExec` is created as a result of [CollapseCodegenStages](#) physical query optimization rule, it is only executed in [executedPlan](#) phase of a query execution (that you can only notice by the `*` star prefix in a plan output).

```

val q = spark.range(9)

// we need executedPlan with WholeStageCodegenExec physical operator "injected"
val plan = q.queryExecution.executedPlan

// Note the star prefix of Range that marks WholeStageCodegenExec
// As a matter of fact, there are two physical operators in play here
// i.e. WholeStageCodegenExec with Range as the child
scala> println(plan.numberedTreeString)
00 *Range (0, 9, step=1, splits=8)

// Let's unwrap Range physical operator
// and access the parent WholeStageCodegenExec
import org.apache.spark.sql.execution.WholeStageCodegenExec
val wsce = plan.asInstanceOf[WholeStageCodegenExec]

// Trigger code generation of the entire query plan tree
val (ctx, code) = wsce.doCodeGen

// CodeFormatter can pretty-print the code
import org.apache.spark.sql.catalyst.expressions.codegen.CodeFormatter
scala> println(CodeFormatter.format(code))
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ /**
 * Codegend pipeline for
 * Range (0, 9, step=1, splits=8)
 */
/* 006 */ final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {
    ...
}

```

When [executed](#), `WholeStageCodegenExec` gives [pipelineTime](#) performance metric.

Table 1. WholeStageCodegenExec's Performance Metrics

Key	Name (in web UI)	Description
<code>pipelineTime</code>	(empty)	Time of how long the whole-stage codegen pipeline has been running (i.e. the elapsed time since the underlying BufferedRowIterator had been created and the internal rows were all consumed).

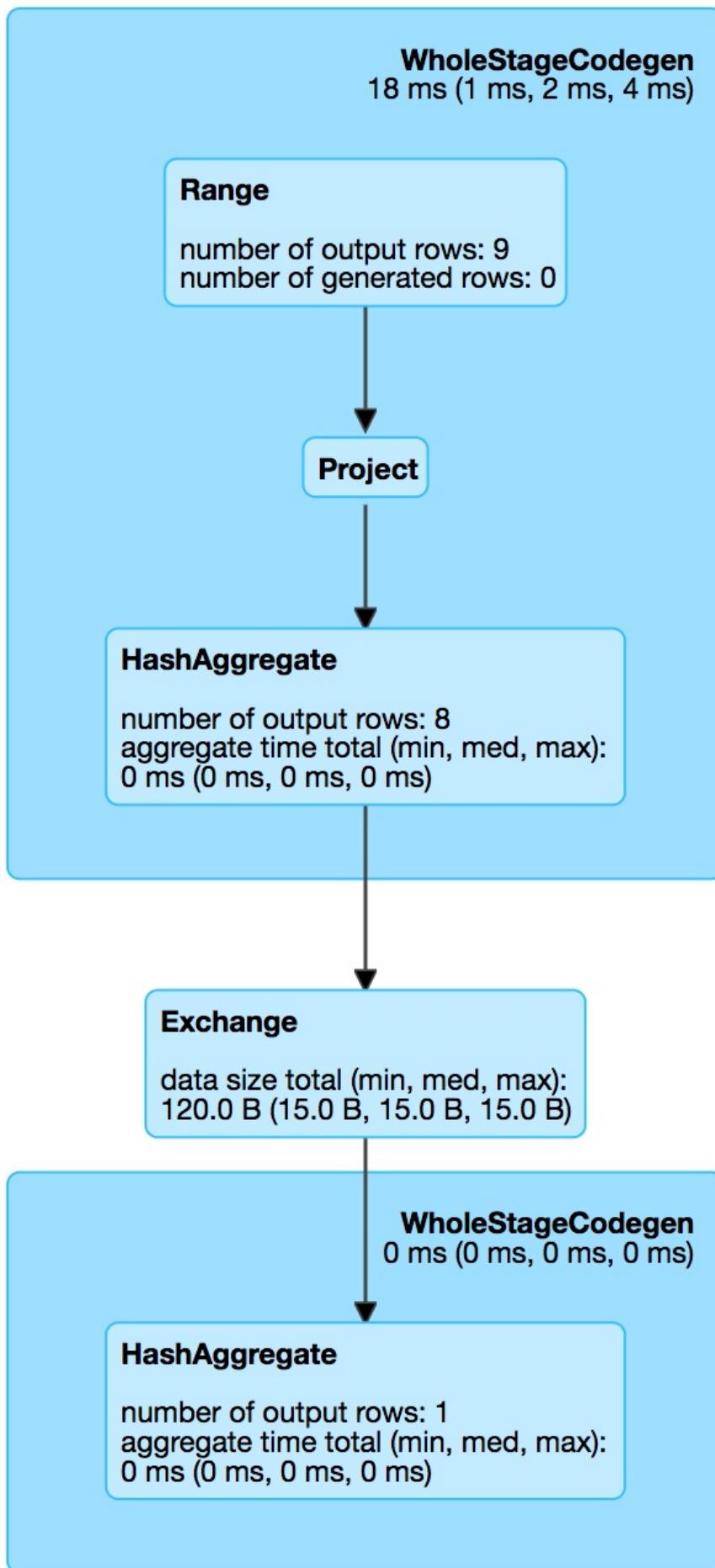


Figure 1. WholeStageCodegenExec in web UI (Details for Query)

Tip	Use <code>explain</code> operator to know the physical plan of a query and find out whether or not <code>WholeStageCodegen</code> is in use.
-----	--

```
val q = spark.range(10).where('id === 4)
// Note the stars in the output that are for codegened operators
scala> q.explain
== Physical Plan ==
*Filter (id#0L = 4)
+- *Range (0, 10, step=1, splits=8)
```

Note	Physical plans that support code generation extend CodegenSupport .
------	---

Enable `DEBUG` logging level for `org.apache.spark.sql.execution.WholeStageCodegenExec` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.WholeStageCodegenExec=DEBUG
```

Refer to [Logging](#).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	<code>doExecute</code> is part of SparkPlan Contract to generate the runtime representation of a structured query as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	---

`doExecute` generates the Java source code for the child physical plan subtree first and uses `CodeGenerator` to compile it right afterwards.

If compilation goes well, `doExecute` branches off per the number of [input RDDs](#).

Note	<code>doExecute</code> only supports up to two input RDDs .
------	---

Caution	FIXME Finish the "success" path
---------	---------------------------------

If the size of the generated codes is greater than `spark.sql.codegen.hugeMethodLimit` (which defaults to `65535`), `doExecute` prints out the following INFO message:

```
Found too long generated codes and JIT optimization might not work: the bytecode size ([maxCodeSize]) is above the limit [spark.sqlcodegen.hugeMethodLimit], and the whole-stage codegen was disabled for this plan (id=[codegenStageId]). To avoid this, you can raise the limit `spark.sqlcodegen.hugeMethodLimit`:
[treeString]
```

In the end, `doExecute` requests the `child` physical operator to `execute` (that triggers physical query planning and generates an `RDD[InternalRow]`) and returns it.

Note

`doExecute` skips requesting the `child` physical operator to `execute` for `FileSourceScanExec` leaf physical operator with `supportsBatch` flag enabled (as `FileSourceScanExec` operator uses `WholeStageCodegenExec` operator when `FileSourceScanExec`).

If compilation fails and `spark.sqlcodegen.fallback` configuration property is enabled, `doExecute` prints out the following WARN message to the logs, requests the `child` physical operator to `execute` and returns it.

```
Whole-stage codegen disabled for plan (id=[codegenStageId]):
[treeString]
```

Generating Java Source Code for Child Physical Plan Subtree— `doCodeGen` Method

```
doCodeGen(): (CodegenContext, CodeAndComment)
```

`doCodeGen` creates a new `CodegenContext` and requests the single `child` physical operator to generate a Java source code for produce code path (with the new `CodegenContext` and the `WholeStageCodegenExec` physical operator itself).

`doCodeGen` adds the new function under the name of `processNext`.

`doCodeGen` generates the class name.

`doCodeGen` generates the final Java source code of the following format:

```

public Object generate(Object[] references) {
    return new [className](references);
}

/**
 * Codegend pipeline for stage (id=[codegenStageId])
 * [treeString]
 */
final class [className] extends BufferedRowIterator {

    private Object[] references;
    private scala.collection.Iterator[] inputs;
    // ctx.declareMutableStates()

    public [className](Object[] references) {
        this.references = references;
    }

    public void init(int index, scala.collection.Iterator[] inputs) {
        partitionIndex = index;
        this.inputs = inputs;
        // ctx.initMutableStates()
        // ctx.initPartition()
    }

    // ctx.emitExtraCode()

    // ctx.declareAddedFunctions()
}

```

Note

`doCodeGen` requires that the single `child` physical operator supports Java code generation.

`doCodeGen` cleans up the generated code (using `CodeFormatter` to `stripExtraNewLines`, `stripOverlappingComments`).

`doCodeGen` prints out the following DEBUG message to the logs:

```

DEBUG WholeStageCodegenExec:
[cleanedSource]

```

In the end, `doCodeGen` returns the `CodegenContext` and the Java source code (as a `CodeAndComment`).

Note

`doCodeGen` is used when:

- `WholeStageCodegenExec` is [executed](#)
- Debugging Query Execution is requested to [display a Java source code generated for a structured query in Whole-Stage Code Generation](#)

Generating Java Source Code for Consume Path in Whole-Stage Code Generation — `doConsume` Method

```
doConsume(ctx: CodegenContext, input: Seq[ExprCode], row: ExprCode): String
```

Note

`doConsume` is part of [CodegenSupport Contract](#) to generate the Java source code for [consume path](#) in Whole-Stage Code Generation.

`doConsume` generates a Java source code that:

1. Takes (from the input `row`) the code to evaluate a Catalyst expression on an input `InternalRow`
2. Takes (from the input `row`) the term for a value of the result of the evaluation
 - i. Adds `.copy()` to the term if [needCopyResult](#) is turned on
3. Wraps the term inside `append()` code block

```
import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext()

import org.apache.spark.sql.catalyst.expressions.codegen.ExprCode
val exprCode = ExprCode(code = "my_code", isNull = "false", value = "my_value")

// wsce defined above, i.e at the top of the page
val consumeCode = wsce.doConsume(ctx, input = Seq(), row = exprCode)
scala> println(consumeCode)
my_code
append(my_value);
```

Generating Class Name — `generatedClassName` Method

```
generatedClassName(): String
```

`generatedClassName` gives a class name per `spark.sql.codegen.useldInClassName` configuration property:

- `GeneratedIteratorForCodegenStage` with the `codegen stage ID` when enabled (`true`)
- `GeneratedIterator` when disabled (`false`)

Note

`generatedClassName` is used exclusively when `WholeStageCodegenExec` unary physical operator is requested to generate the Java source code for the child physical plan subtree.

isToManyFields Object Method

`isToManyFields(conf: SQLConf, dataType: DataType): Boolean`

`isToManyFields ...FIXME`

Note

`isToManyFields` is used when...FIXME

WriteToDataSourceV2Exec Physical Operator

`WriteToDataSourceV2Exec` is a [physical operator](#) that represents an [AppendData](#) logical operator (and a deprecated [WriteToDataSourceV2](#) logical operator) at execution time.

`WriteToDataSourceV2Exec` is [created](#) exclusively when [DataSourceV2Strategy](#) execution planning strategy is requested to plan an [AppendData](#) logical operator (and a deprecated [WriteToDataSourceV2](#)).

Note	Although WriteToDataSourceV2 logical operator is deprecated since Spark SQL 2.4.0 (for AppendData logical operator), the <code>AppendData</code> logical operator is currently used in tests only. That makes <code>writeToDataSourcev2</code> logical operator still relevant.
------	---

`WriteToDataSourceV2Exec` takes the following to be created:

- [DataSourceWriter](#)
- Child [physical plan](#)

When requested for the [child operators](#), `WriteToDataSourceV2Exec` gives the one [child physical plan](#).

When requested for the [output attributes](#), `WriteToDataSourceV2Exec` gives no attributes (an empty collection).

Tip	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exec</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.datasources.v2.WriteToDataSourceV2Exe</pre> <p>Refer to Logging.</p>
-----	--

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of [SparkPlan Contract](#) to generate the runtime representation of a structured query as a distributed computation over [internal binary rows](#) on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` requests the [DataSourceWriter](#) to create a [DataWriterFactory](#) for the writing task.

`doExecute` requests the [DataSourceWriter](#) to use a [CommitCoordinator](#) or not.

`doExecute` requests the [child physical plan](#) to [execute](#) (that triggers physical query planning and in the end generates an `RDD` of [internal binary rows](#)).

`doExecute` prints out the following INFO message to the logs:

```
Start processing data source writer: [writer]. The input RDD has [length] partitions.
```

`doExecute` requests the [SparkContext](#) to run a Spark job with the following:

- The `RDD[InternalRow]` of the [child physical plan](#)
- A partition processing function that requests the `DataWritingSparkTask` object to [run](#) the writing task (of the [DataSourceWriter](#)) with or with no commit coordinator
- A result handler function that records the result `writerCommitMessage` from a successful data writer and requests the [DataSourceWriter](#) to [handle the commit message](#) (which does nothing by default)

`doExecute` prints out the following INFO message to the logs:

```
Data source writer [writer] is committing.
```

`doExecute` requests the [DataSourceWriter](#) to [commit](#) (passing on with the commit messages).

In the end, `doExecute` prints out the following INFO message to the logs:

```
Data source writer [writer] committed.
```

In case of any error (`Throwable`), `doExecute ...FIXME`

AliasViewChild Logical Analysis Rule

`AliasViewChild` is a logical analysis rule that transforms a logical query plan with `View` unary logical operators and adds `Project` logical operator (possibly with `Alias` expressions) when the outputs of a view and the underlying table do not match (and therefore require aliasing and projection).

`AliasViewChild` is part of the `View` once-executed batch in the standard batches of the `Analyzer`.

`AliasViewChild` is simply a `Catalyst rule` for transforming `logical plans`, i.e.
`Rule[LogicalPlan]` .

`AliasViewChild` takes a `SQLConf` when created.

```

// Sample view for the demo
// The order of the column names do not match
// In View: name, id
// In VALUES: id, name
sql("""
    CREATE OR REPLACE VIEW v (name COMMENT 'First name only', id COMMENT 'Identifier') C
OMMENT 'Permanent view'
    AS VALUES (1, 'Jacek'), (2, 'Agata') AS t1(id, name)
""")

scala> :type spark
org.apache.spark.sql.SparkSession

val q = spark.table("v")

val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedRelation `v`

// Resolve UnresolvedRelation first
// since AliasViewChild work with View operators only
import spark.sessionState.analyzer.ResolveRelations
val resolvedPlan = ResolveRelations(plan)
scala> println(resolvedPlan.numberedTreeString)
00 SubqueryAlias v
01 +- View (`default`.`v`, [name#32,id#33])
02     +- SubqueryAlias t1
03         +- LocalRelation [id#34, name#35]

scala> :type spark.sessionState.conf
org.apache.spark.sql.internal.SQLConf

import org.apache.spark.sql.catalyst.analysis.AliasViewChild
val rule = AliasViewChild(spark.sessionState.conf)

// Notice that resolvedPlan is used (not plan)
val planAfterAliasViewChild = rule(resolvedPlan)
scala> println(planAfterAliasViewChild.numberedTreeString)
00 SubqueryAlias v
01 +- View (`default`.`v`, [name#32,id#33])
02     +- Project [cast(id#34 as int) AS name#32, cast(name#35 as string) AS id#33]
03         +- SubqueryAlias t1
04             +- LocalRelation [id#34, name#35]

```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

CleanupAliases Logical Analysis Rule

`CleanupAliases` is a [logical analysis rule](#) that transforms a logical query plan with...FIXME

`CleanupAliases` is part of the [Cleanup](#) fixed-point batch in the standard batches of the [Analyzer](#).

`CleanupAliases` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan]` .

```
// FIXME: DEMO
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` ...FIXME

DataSourceAnalysis PostHoc Logical Resolution Rule

`DataSourceAnalysis` is a [posthoc logical resolution rule](#) that the [default](#) and [Hive-specific](#) logical query plan analyzers use to [FIXME](#).

Table 1. `DataSourceAnalysis`'s Logical Resolutions (Conversions)

Source Operator	Target Operator	Description
<code>CreateTable</code> (<code>isDatasourceTable</code> + no query)	<code>CreateDataSourceTableCommand</code>	
<code>CreateTable</code> (<code>isDatasourceTable</code> + a resolved query)	<code>CreateDataSourceTableAsSelectCommand</code>	
<code>InsertIntoTable</code> with <code>InsertableRelation</code>	<code>InsertIntoDataSourceCommand</code>	
<code>InsertIntoDir</code> (non- hive provider)	<code>InsertIntoDataSourceDirCommand</code>	
<code>InsertIntoTable</code> with <code>HadoopFsRelation</code>	<code>InsertIntoHadoopFsRelationCommand</code>	

Technically, `DataSourceAnalysis` is a [Catalyst rule](#) for transforming [logical plans](#), i.e.

```
Rule[LogicalPlan] .
```

```
// FIXME Example of DataSourceAnalysis
import org.apache.spark.sql.execution.datasources.DataSourceAnalysis
val rule = DataSourceAnalysis(spark.sessionState.conf)

val plan = FIXME

rule(plan)
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

DetermineTableStats Logical PostHoc Resolution Rule — Computing Total Size Table Statistic for HiveTableRelations

`DetermineTableStats` is a [logical posthoc resolution rule](#) that the [Hive-specific logical query plan analyzer](#) uses to [compute total size table statistic for HiveTableRelations with no statistics](#).

Technically, `DetermineTableStats` is a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of Rule Contract to apply a rule to a logical plan (aka <i>execute a rule</i>).
------	---

```
apply ...FIXME
```

ExtractWindowExpressions Logical Resolution Rule

`ExtractWindowExpressions` is a [logical resolution rule](#) that transforms a logical query plan and replaces (extracts) [WindowExpression](#) expressions with [Window](#) logical operators.

`ExtractWindowExpressions` is part of the [Resolution](#) fixed-point batch in the standard batches of the [Analyzer](#).

`ExtractWindowExpressions` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.
`Rule[LogicalPlan]`.

Note	<code>ExtractWindowExpressions</code> is a Scala object inside Analyzer class (so you have to create an instance of the <code>Analyzer</code> class to access it or simply use SessionState).
------	--

```
import spark.sessionState.analyzer.ExtractWindowExpressions

// Example 1: Filter + Aggregate with WindowExpressions in aggregateExprs
val q = ???
val plan = q.queryExecution.logical
val afterExtractWindowExpressions = ExtractWindowExpressions(plan)

// Example 2: Aggregate with WindowExpressions in aggregateExprs
val q = ???
val plan = q.queryExecution.logical
val afterExtractWindowExpressions = ExtractWindowExpressions(plan)

// Example 3: Project with WindowExpressions in projectList
val q = ???
val plan = q.queryExecution.logical
val afterExtractWindowExpressions = ExtractWindowExpressions(plan)
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (apply) a rule on a TreeNode (e.g. LogicalPlan).
------	--

`apply` transforms the logical operators downwards in the input [logical plan](#) as follows:

- For `Filter` unary operators with `Aggregate` operator (as the `child`) that `has a window function` in the `aggregateExpressions`, apply ...FIXME
- For `Aggregate` logical operators that `have a window function` in the `aggregateExpressions`, apply ...FIXME
- For `Project` logical operators that `have a window function` in the `projectList`, apply ...FIXME

hasWindowFunction Internal Method

```
hasWindowFunction(projectList: Seq[NamedExpression]): Boolean (1)
hasWindowFunction(expr: NamedExpression): Boolean
```

1. Executes the other `hasWindowFunction` on every `NamedExpression` in the `projectList`

`hasWindowFunction` is positive (`true`) when the input `expr named expression` is a `WindowExpression` expression. Otherwise, `hasWindowFunction` is negative (`false`).

Note	<code>hasWindowFunction</code> is used when <code>ExtractWindowExpressions</code> logical resolution rule is requested to <code>extract</code> and <code>execute</code> .
------	---

extract Internal Method

```
extract(expressions: Seq[NamedExpression]): (Seq[NamedExpression], Seq[NamedExpression])
```

`extract` ...FIXME

Note	<code>extract</code> is used exclusively when <code>ExtractWindowExpressions</code> logical resolution rule is <code>executed</code> .
------	--

Adding Project and Window Logical Operators to Logical Plan — addWindow Internal Method

```
addWindow(
  expressionsWithWindowFunctions: Seq[NamedExpression],
  child: LogicalPlan): LogicalPlan
```

`addWindow` adds a `Project` logical operator with one or more `Window` logical operators (for every `WindowExpression` in the input `named expressions`) to the input `logical plan`.

Internally, `addWindow` ...FIXME

Note

`addWindow` is used exclusively when `ExtractWindowExpressions` logical resolution rule is [executed](#).

FindDataSourceTable Logical Evaluation Rule for Resolving UnresolvedCatalogRelations

`FindDataSourceTable` is a [Catalyst rule](#) that the [default](#) and [Hive-specific](#) logical query plan analyzers use for [resolving UnresolvedCatalogRelations](#) in a logical plan for the following cases:

- [InsertIntoTables](#) with `UnresolvedCatalogRelation` (for datasource and hive tables)
- "Standalone" `UnresolvedCatalogRelations`

Note	<code>UnresolvedCatalogRelation</code> leaf logical operator is a placeholder that ResolveRelations logical evaluation rule adds to a logical plan while resolving UnresolvedRelations leaf logical operators.
------	--

`FindDataSourceTable` is part of [additional rules](#) in `Resolution` fixed-point batch of rules.

```

scala> :type spark
org.apache.spark.sql.SparkSession

// Example: InsertIntoTable with UnresolvedCatalogRelation
// Drop tables to make the example reproducible
val db = spark.catalog.currentDatabase
Seq("t1", "t2").foreach { t =>
  spark.sharedState.externalCatalog.dropTable(db, t, ignoreIfNotExists = true, purge = true)
}

// Create tables
sql("CREATE TABLE t1 (id LONG) USING parquet")
sql("CREATE TABLE t2 (id LONG) USING orc")

import org.apache.spark.sql.catalyst.dsl.plans._
val plan = table("t1").insertInto(tableName = "t2", overwrite = true)
scala> println(plan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- 'UnresolvedRelation `t1`

// Transform the logical plan with ResolveRelations logical rule first
// so UnresolvedRelations become UnresolvedCatalogRelations
import spark.sessionState.analyzer.ResolveRelations
val planWithUnresolvedCatalogRelations = ResolveRelations(plan)
scala> println(planWithUnresolvedCatalogRelations.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- 'SubqueryAlias t1
02   +- 'UnresolvedCatalogRelation `default`.`t1`, org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe

// Let's resolve UnresolvedCatalogRelations then
import org.apache.spark.sql.execution.datasources.FindDataSourceTable
val r = new FindDataSourceTable(spark)
val tablesResolvedPlan = r(planWithUnresolvedCatalogRelations)
// FIXME Why is t2 not resolved?!
scala> println(tablesResolvedPlan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- SubqueryAlias t1
02   +- Relation[id#10L] parquet

```

Applying FindDataSourceTable Rule to Logical Plan (and Resolving UnresolvedCatalogRelations in Logical Plan)

— apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of Rule Contract to apply a rule to a logical plan.
------	--

`apply ...FIXME`

readHiveTable Internal Method

```
readHiveTable(table: CatalogTable): LogicalPlan
```

`readHiveTable` simply creates a `HiveTableRelation` for the input `CatalogTable`.

Note	<code>readHiveTable</code> is used when <code>FindDataSourceTable</code> is requested to resolving <code>UnresolvedCatalogRelations</code> in a logical plan.
------	---

readDataSourceTable Internal Method

```
readDataSourceTable(table: CatalogTable): LogicalPlan
```

`readDataSourceTable ...FIXME`

Note	<code>readDataSourceTable</code> is used exclusively when <code>FindDataSourceTable</code> logical evaluation rule is <code>executed</code> (for data source tables).
------	---

HandleNullInputsForUDF Logical Evaluation Rule

`HandleNullInputsForUDF` is a logical evaluation rule (i.e. `Rule[LogicalPlan]`) that Spark SQL's [logical query analyzer](#) uses to...FIXME

HiveAnalysis PostHoc Logical Resolution Rule

`HiveAnalysis` is a [logical posthoc resolution rule](#) that the [Hive-specific logical query plan analyzer](#) uses to [FIXME](#).

Technically, `HiveAnalysis` is a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

```
// FIXME Example of HiveAnalysis
```

Applying HiveAnalysis Rule to Logical Plan (Executing HiveAnalysis) — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

`apply ...FIXME`

InConversion Type Coercion Logical Rule

`InConversion` is a [type coercion logical rule](#) that `coerceTypes` in a [logical plan](#).

Coercing Types in Logical Plan — `coerceTypes` Method

```
coerceTypes(plan: LogicalPlan): LogicalPlan
```

Note

`coerceTypes` is part of the [TypeCoercionRule Contract](#) to coerce types in a [logical plan](#).

`coerceTypes` ...FIXME

LookupFunctions Logical Rule — Checking Whether UnresolvedFunctions Are Resolvable

`LookupFunctions` is a logical rule that the [logical query plan analyzer](#) uses to make sure that [UnresolvedFunction expressions can be resolved](#) in an entire logical query plan.

`LookupFunctions` is similar to [ResolveFunctions](#) logical resolution rule, but it is [ResolveFunctions](#) to resolve [UnresolvedFunction](#) expressions while `LookupFunctions` is just a sanity check that a future resolution is possible if tried.

Technically, `LookupFunctions` is just a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

Note `LookupFunctions` does not however transform a logical plan.

`LookupFunctions` is part of [Simple Sanity Check](#) one-off batch of rules.

Note `LookupFunctions` is a Scala object inside [Analyzer](#) class.

```
// Using Catalyst DSL to create a logical plan with an unregistered function
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")

import org.apache.spark.sql.catalyst.dsl.expressions._
val f1 = 'f1.function()

val plan = t1.select(f1)
scala> println(plan.numberedTreeString)
00 'Project [unresolvedalias('f1(), None)]
01 +- 'UnresolvedRelation `t1`

// Make sure the function f1 does not exist
import org.apache.spark.sql.catalyst.FunctionIdentifier
spark.sessionState.catalog.dropFunction(FunctionIdentifier("f1"), ignoreIfNotExists =
true)

assert(spark.catalog.functionExists("f1") == false)

import spark.sessionState.analyzer.LookupFunctions
scala> LookupFunctions(plan)
org.apache.spark.sql.AnalysisException: Undefined function: 'f1'. This function is nei
ther a registered temporary function nor a permanent function registered in the databa
se 'default'.;
    at org.apache.spark.sql.catalyst.analysis.Analyzer$LookupFunctions$$anonfun$apply$15
    $$anonfun$applyOrElse$49.apply(Analyzer.scala:1198)
    at org.apache.spark.sql.catalyst.analysis.Analyzer$LookupFunctions$$anonfun$apply$15
    $$anonfun$applyOrElse$49.apply(Analyzer.scala:1198)
    at org.apache.spark.sql.catalyst.analysis.package$.withPosition(package.scala:48)
    at org.apache.spark.sql.catalyst.analysis.Analyzer$LookupFunctions$$anonfun$apply$15.
applyOrElse(Analyzer.scala:1197)
    at org.apache.spark.sql.catalyst.analysis.Analyzer$LookupFunctions$$anonfun$apply$15.
applyOrElse(Analyzer.scala:1195)
```

Applying LookupFunctions to Logical Plan — apply Method

`apply(plan: LogicalPlan): LogicalPlan`

Note

`apply` is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

`apply` finds all [UnresolvedFunction](#) expressions (in every logical operator in the input [logical plan](#)) and requests the [SessionCatalog](#) to [check if their functions exist](#).

`apply` does nothing if a function exists or reports a [NoSuchFunctionException](#) (that fails logical analysis).

```
Undefined function: '[func]'. This function is neither a registered temporary function  
nor a permanent function registered in the database '[db]'.
```

PreprocessTableCreation PostHoc Logical Resolution Rule

`PreprocessTableCreation` is a [posthoc logical resolution rule](#) that [resolves a logical query plan](#) with [CreateTable](#) logical operators.

`PreprocessTableCreation` is part of the [Post-Hoc Resolution](#) once-executed batch of the [Hive-specific](#) and the [default](#) logical analyzers.

`PreprocessTableCreation` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.
`Rule[LogicalPlan]`.

`PreprocessTableCreation` takes a [SparkSession](#) when created.

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` ...FIXME

PreWriteCheck Extended Analysis Check

`PreWriteCheck` is an **extended analysis check** that verifies correctness of a [logical query plan](#) with regard to [InsertIntoTable](#) unary logical operator (right before analysis can be considered complete).

`PreWriteCheck` is part of the [extended analysis check rules](#) of the logical [Analyzer](#) in [BaseSessionStateBuilder](#) and [HiveSessionStateBuilder](#).

`PreWriteCheck` is simply a [function](#) of [LogicalPlan](#) that...FIXME

Executing Function — `apply` Method

```
apply(plan: LogicalPlan): Unit
```

Note	<code>apply</code> is part of Scala's scala.Function1 contract to create a function of one parameter.
------	---

`apply` [traverses](#) the input [logical query plan](#) and finds [InsertIntoTable](#) unary logical operators.

- For an [InsertIntoTable](#) with a [LogicalRelation](#)...FIXME
- For any [InsertIntoTable](#), `apply` throws a [AnalysisException](#) if the [logical plan for the table to insert into](#) is neither a [LeafNode](#) nor one of the following leaf logical operators: [Range](#), [OneRowRelation](#), [LocalRelation](#).

```
Inserting into an RDD-based table is not allowed.
```

RelationConversions Logical PostHoc Evaluation Rule — Converting Hive Tables

`RelationConversions` is a [logical posthoc resolution rule](#) that the [Hive-specific logical query plan analyzer](#) uses to [convert a Hive table...FIXME](#).

Note	A Hive table is when the <code>provider</code> is <code>hive</code> in <code>table metadata</code> .
------	--

Caution	FIXME Show example of a hive table, e.g. <code>spark.table(...)</code>
---------	--

`RelationConversions` is [created](#) exclusively when the [Hive-specific logical query plan analyzer](#) is created.

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (apply) a rule on a TreeNode (e.g. LogicalPlan).
------	--

`apply` traverses the input [logical plan](#) looking for a [InsertIntoTable](#) with [HiveTableRelation](#) logical operators or [HiveTableRelation](#) logical operator alone.

For a [InsertIntoTable](#) with [non-partitioned](#) [HiveTableRelation](#) relation (that can be [converted](#)) `apply` [converts](#) the [HiveTableRelation](#) to a [LogicalRelation](#).

For a [HiveTableRelation](#) logical operator alone `apply` ...FIXME

Creating RelationConversions Instance

`RelationConversions` takes the following when created:

- [SQLConf](#)
- [Hive-specific session catalog](#)

Does Table Use Parquet or ORC SerDe? — `isConvertible` Internal Method

```
isConvertible(relation: HiveTableRelation): Boolean
```

`isConvertible` is positive when the input `HiveTableRelation` is a parquet or ORC table (and corresponding SQL properties are enabled).

Internally, `isConvertible` takes the Hive SerDe of the table (from `table metadata`) if available or assumes no SerDe.

`isConvertible` is turned on when either condition holds:

- The Hive SerDe is `parquet` (aka *parquet table*) and `spark.sql.hive.convertMetastoreParquet` configuration property is enabled (which is by default)
- The Hive SerDe is `orc` (aka *orc table*) and `spark.sql.hive.convertMetastoreOrc` internal configuration property is enabled (which is by default)

Note

`isConvertible` is used when `RelationConversions` is [executed](#).

Converting `HiveTableRelation` to `LogicalRelation`

— `convert` Internal Method

```
convert(relation: HiveTableRelation): LogicalRelation
```

`convert` takes SerDe of (the storage of) the input `HiveTableRelation` and converts `HiveTableRelation` to `LogicalRelation`, i.e.

1. For `parquet` `serde`, `convert` adds `mergeSchema` option being the value of `spark.sql.hive.convertMetastoreParquet.mergeSchema` configuration property (disabled by default) and requests `HiveMetastoreCatalog` to `convertToLogicalRelation` (with `ParquetFileFormat` as `fileFormatClass`).

For non- `parquet` `serde`, `convert` assumes ORC format.

- When `spark.sql.orc.impl` configuration property is `native` (default) `convert` requests `HiveMetastoreCatalog` to `convertToLogicalRelation` (with `org.apache.spark.sql.execution.datasources.orc.OrcFileFormat` as `fileFormatClass`).
- Otherwise, `convert` requests `HiveMetastoreCatalog` to `convertToLogicalRelation` (with `org.apache.spark.sql.hive.orc.OrcFileFormat` as `fileFormatClass`).

Note

`convert` uses `HiveSessionCatalog` to access the `HiveMetastoreCatalog`.

Note

`convert` is used when `RelationConversions` logical evaluation rule does the following transformations:

- Transforms a [InsertIntoTable](#) with `HiveTableRelation` with a Hive table (i.e. with `hive` provider) that is not partitioned and uses `parquet` or `orc` data storage format
- Transforms a [HiveTableRelation](#) with a Hive table (i.e. with `hive` provider) that uses `parquet` or `orc` data storage format

ResolveAliases Logical Resolution Rule

`ResolveAliases` is a logical resolution rule that the [logical query plan analyzer](#) uses to [FIXME](#) in an entire logical query plan.

Technically, `ResolveAliases` is just a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`ResolveAliases` is part of [Resolution](#) fixed-point batch of rules.

Note	<code>ResolveAliases</code> is a Scala object inside Analyzer class.
------	--

```
import spark.sessionState.analyzer.ResolveAliases  
  
// FIXME Using ResolveAliases rule
```

Applying `ResolveAliases` to Logical Plan — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of Rule Contract to apply a rule to a logical plan .
------	---

`apply` ...FIXME

assignAliases Internal Method

```
assignAliases(exprs: Seq[NamedExpression]): Seq[NamedExpression]
```

`assignAliases` ...FIXME

Note	<code>assignAliases</code> is used when...FIXME
------	---

ResolveBroadcastHints Logical Resolution Rule — Resolving UnresolvedHint Operators with BROADCAST, BROADCASTJOIN and MAPJOIN Hint Names

`ResolveBroadcastHints` is a logical resolution rule that the [Spark Analyzer](#) uses to [resolve UnresolvedHint logical operators](#) with `BROADCAST`, `BROADCASTJOIN` or `MAPJOIN` hints (case-insensitive) to [ResolvedHint](#) operators.

Technically, `ResolveBroadcastHints` is a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`ResolveBroadcastHints` is part of [Hints](#) fixed-point batch of rules (that is executed before any other rule).

`ResolveBroadcastHints` takes a [SQLConf](#) when created.

```
// Use Catalyst DSL to create a logical plan
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = table("t1").join(table("t2")).hint(name = "broadcast", "t1", "table2")
scala> println(plan.numberedTreeString)
00 'UnresolvedHint broadcast, [t1, table2]
01 +- 'Join Inner
02   :- 'UnresolvedRelation `t1`
03   +- 'UnresolvedRelation `t2`

import org.apache.spark.sql.catalyst.analysis.ResolveHints.ResolveBroadcastHints
val resolver = new ResolveBroadcastHints(spark.sessionState.conf)
val analyzedPlan = resolver(plan)
scala> println(analyzedPlan.numberedTreeString)
00 'Join Inner
01 :- 'ResolvedHint (broadcast)
02 : +- 'UnresolvedRelation `t1`
03 +- 'UnresolvedRelation `t2`
```

Resolving UnresolvedHint with BROADCAST, BROADCASTJOIN or MAPJOIN Hint Names (Applying `ResolveBroadcastHints` to Logical Plan) — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of Rule Contract to apply a rule to a logical plan.
------	--

`apply` transforms `UnresolvedHint` operators into `ResolvedHint` for the hint names as `BROADCAST`, `BROADCASTJOIN` or `MAPJOIN` (case-insensitive).

For `UnresolvedHints` with no parameters, `apply` marks the entire child logical plan as eligible for broadcast, i.e. creates a `ResolvedHint` with the child operator and `HintInfo` with `broadcast` flag on.

For `UnresolvedHints` with parameters defined, `apply` considers the parameters the names of the tables to apply broadcast hint to.

Note	The table names can be of <code>String</code> or <code>UnresolvedAttribute</code> types.
------	--

`apply` reports an `AnalysisException` for the parameters that are not of `String` or `UnresolvedAttribute` types.

```
org.apache.spark.sql.AnalysisException: Broadcast hint parameter should be an identifier or string but was [unsupported] ([className]
```

```

// Use Catalyst DSL to create a logical plan
import org.apache.spark.sql.catalyst.dsl.plans._

// !!! IT WON'T WORK !!!
// 1 is not a table name or of type `UnresolvedAttribute`
val plan = table("t1").hint(name = "broadcast", 1)

scala> println(plan.numberedTreeString)
00 'UnresolvedHint broadcast, [1]
01 +- 'UnresolvedRelation `t1`

// Resolve hints
import org.apache.spark.sql.catalyst.analysis.ResolveHints
val broadcastHintResolver = new ResolveHints.ResolveBroadcastHints(spark.sessionState.conf)
scala> broadcastHintResolver(plan)
org.apache.spark.sql.AnalysisException: Broadcast hint parameter should be an identifier or string but was 1 (class java.lang.Integer;
  at org.apache.spark.sql.catalyst.analysis.ResolveHints$ResolveBroadcastHints$$anonfun$apply$1$$anonfun$applyOrElse$1.apply(ResolveHints.scala:98)
  at org.apache.spark.sql.catalyst.analysis.ResolveHints$ResolveBroadcastHints$$anonfun$apply$1$$anonfun$applyOrElse$1.apply(ResolveHints.scala:95)
  at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
  at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
  at scala.collection.IndexedSeqOptimized$class.foreach(IndexedSeqOptimized.scala:33)
  at scala.collection.mutable.WrappedArray.foreach(WrappedArray.scala:35)
  at scala.collection.TraversableLike$class.map(TraversableLike.scala:234)
  at scala.collection.AbstractTraversable.map(Traversable.scala:104)
  at org.apache.spark.sql.catalyst.analysis.ResolveHints$ResolveBroadcastHints$$anonfun$apply$1$.applyOrElse(ResolveHints.scala:95)
  at org.apache.spark.sql.catalyst.analysis.ResolveHints$ResolveBroadcastHints$$anonfun$apply$1$.applyOrElse(ResolveHints.scala:88)
  at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$transformUp$1.apply(TreeNode.scala:289)
  at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$transformUp$1.apply(TreeNode.scala:289)
  at org.apache.spark.sql.catalyst.trees.CurrentOrigin$.withOrigin(TreeNode.scala:70)
  at org.apache.spark.sql.catalyst.trees.TreeNode.transformUp(TreeNode.scala:288)
  at org.apache.spark.sql.catalyst.analysis.ResolveHints$ResolveBroadcastHints.apply(ResolveHints.scala:88)
  ...
  ... 51 elided

```

applyBroadcastHint Internal Method

```
applyBroadcastHint(plan: LogicalPlan, toBroadcast: Set[String]): LogicalPlan
```

applyBroadcastHint ...FIXME

Note

`applyBroadcastHint` is used exclusively when `ResolveBroadcastHints` is requested to [execute](#).

ResolveCoalesceHints Logical Resolution Rule — Resolving UnresolvedHint Operators with COALESCE and REPARTITION Hints

`ResolveCoalesceHints` is a logical resolution rule that the [Spark Analyzer](#) uses to [resolve](#) [UnresolvedHint logical operators](#) with `COALESCE` or `REPARTITION` hints (case-insensitive) to [ResolvedHint](#) operators.

`COALESCE` or `REPARTITION` hints expect a partition number as the only parameter.

Technically, `ResolveCoalesceHints` is a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`ResolveCoalesceHints` is part of [Hints](#) fixed-point batch of rules (that is executed before any other rule).

`ResolveCoalesceHints` takes no input parameters when created.

Example: Using COALESCE Hint

```
// Use Catalyst DSL to create a logical plan
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = table("t1").hint(name = "COALESCE", 3)
scala> println(plan.numberedTreeString)
00 'UnresolvedHint COALESCE, [3]
01 +- 'UnresolvedRelation `t1`

import org.apache.spark.sql.catalyst.analysis.ResolveHints.ResolveCoalesceHints
val analyzedPlan = ResolveCoalesceHints(plan)
scala> println(analyzedPlan.numberedTreeString)
00 'Repartition 3, false
01 +- 'UnresolvedRelation `t1`
```

Example: Using REPARTITION Hint

```
// Use Catalyst DSL to create a logical plan
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = table("t1").hint(name = "REPARTITION", 3)
scala> println(plan.numberedTreeString)
00 'UnresolvedHint REPARTITION, [3]
01 +- 'UnresolvedRelation `t1`

import org.apache.spark.sql.catalyst.analysis.ResolveHints.ResolveCoalesceHints
val analyzedPlan = ResolveCoalesceHints(plan)
scala> println(analyzedPlan.numberedTreeString)
00 'Repartition 3, true
01 +- 'UnresolvedRelation `t1`
```

Example: Using COALESCE Hint in SQL

```
val q = sql("SELECT /*+ COALESCE(10) */ * FROM VALUES 1 t(id)")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedHint COALESCE, [10]
01 +- 'Project [*]
02   +- 'SubqueryAlias `t`
03     +- 'UnresolvedInlineTable [id], [List(1)]

import org.apache.spark.sql.catalyst.analysis.ResolveHints.ResolveCoalesceHints
val analyzedPlan = ResolveCoalesceHints(plan)
scala> println(analyzedPlan.numberedTreeString)
00 'Repartition 10, false
01 +- 'Project [*]
02   +- 'SubqueryAlias `t`
03     +- 'UnresolvedInlineTable [id], [List(1)]
```

ResolveCreateNamedStruct Logical Resolution Rule — Resolving NamePlaceholders In CreateNamedStruct Expressions

`ResolveCreateNamedStruct` is a logical resolution rule that replaces NamePlaceholders with Literals for the names in `CreateNamedStruct` expressions in an entire logical query plan.

`ResolveCreateNamedStruct` is part of the [Resolution](#) fixed-point batch in the standard batches of the [Analyzer](#).

`ResolveCreateNamedStruct` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

val q = spark.range(1).select(struct($"id"))
val logicalPlan = q.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Project [unresolvedalias(named_struct(named_placeholder, 'id), None)]
01 +- AnalysisBarrier
02     +- Range (0, 1, step=1, splits=Some(8))

// Let's resolve references first
import spark.sessionState.analyzer.ResolveReferences
val planWithRefsResolved = ResolveReferences(logicalPlan)

import org.apache.spark.sql.catalyst.analysis.ResolveCreateNamedStruct
val afterResolveCreateNamedStruct = ResolveCreateNamedStruct(planWithRefsResolved)
scala> println(afterResolveCreateNamedStruct.numberedTreeString)
00 'Project [unresolvedalias(named_struct(id, id#4L), None)]
01 +- AnalysisBarrier
02     +- Range (0, 1, step=1, splits=Some(8))
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (<code>apply</code>) a rule on a TreeNode (e.g. LogicalPlan).
------	---

`apply` traverses all Catalyst expressions (in the input LogicalPlan) that are `CreateNamedStruct` expressions which are not resolved yet and replaces `NamePlaceholders` with `Literal` expressions.

In other words, `apply` finds unresolved `CreateNamedStruct` expressions with `NamePlaceholder` expressions in the `children` and replaces them with the `name` of corresponding `NamedExpression`, but only if the `NamedExpression` is resolved.

In the end, `apply` creates a `CreateNamedStruct` with new children.

ResolveFunctions Logical Resolution Rule — Resolving grouping_id UnresolvedAttribute, UnresolvedGenerator And UnresolvedFunction Expressions

`ResolveFunctions` is a logical resolution rule that the [logical query plan analyzer](#) uses to resolve `grouping_id` [UnresolvedAttribute](#), [UnresolvedGenerator](#) and [UnresolvedFunction](#) expressions in an entire logical query plan.

Technically, `ResolveReferences` is just a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`ResolveFunctions` is part of [Resolution](#) fixed-point batch of rules.

Note

`ResolveFunctions` is a Scala object inside [Analyzer](#) class.

```
import spark.sessionState.analyzer.ResolveFunctions

// Example: UnresolvedAttribute with VirtualColumn.hiveGroupingIdName (grouping_id) =
> Alias
import org.apache.spark.sql.catalyst.expressions.VirtualColumn
import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
val groupingIdAttr = UnresolvedAttribute(VirtualColumn.hiveGroupingIdName)
scala> println(groupingIdAttr.sql)
`grouping_id`


// Using Catalyst DSL to create a logical plan with grouping_id
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")
val plan = t1.select(groupingIdAttr)
scala> println(plan.numberedTreeString)
00 'Project ['grouping_id]
01 +- 'UnresolvedRelation `t1`


val resolvedPlan = ResolveFunctions(plan)
scala> println(resolvedPlan.numberedTreeString)
00 'Project [grouping_id() AS grouping_id#0]
01 +- 'UnresolvedRelation `t1`


import org.apache.spark.sql.catalyst.expressions.Alias
val alias = resolvedPlan.expressions.head.asInstanceOf[Alias]
scala> println(alias.sql)
grouping_id() AS `grouping_id`


// Example: UnresolvedGenerator => a) Generator or b) analysis failure
// Register a function so a function resolution works
```

```

import org.apache.spark.sql.catalyst.FunctionIdentifier
import org.apache.spark.sql.catalyst.catalog.CatalogFunction
val f1 = CatalogFunction(FunctionIdentifier(funcName = "f1"), "java.lang.String", resources = Nil)
import org.apache.spark.sql.catalyst.expressions.{Expression, Stack}
// FIXME What happens when looking up a function with the functionBuilder None in registerFunction?
// Using Stack as ResolveFunctions requires that the function to be resolved is a Generator
// You could roll your own, but that's a demo, isn't it? (don't get too carried away)
spark.sessionState.catalog.registerFunction(
  funcDefinition = f1,
  overrideIfExists = true,
  functionBuilder = Some((children: Seq[Expression]) => Stack(children = Nil)))

import org.apache.spark.sql.catalyst.analysis.UnresolvedGenerator
import org.apache.spark.sql.catalyst.FunctionIdentifier
val ungen = UnresolvedGenerator(name = FunctionIdentifier("f1"), children = Seq.empty)
val plan = t1.select(ungen)
scala> println(plan.numberedTreeString)
00 'Project [unresolvedalias('f1(), None)]
01 +- 'UnresolvedRelation `t1`

val resolvedPlan = ResolveFunctions(plan)
scala> println(resolvedPlan.numberedTreeString)
00 'Project [unresolvedalias(stack(), None)]
01 +- 'UnresolvedRelation `t1`

CAUTION: FIXME

// Example: UnresolvedFunction => a) AggregateWindowFunction with(out) isDistinct, b)
AggregateFunction, c) other with(out) isDistinct
val plan = ???
val resolvedPlan = ResolveFunctions(plan)

```

Resolving grouping_id UnresolvedAttribute, UnresolvedGenerator and UnresolvedFunction Expressions In Entire Query Plan (Applying ResolveFunctions to Logical Plan)— apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

`apply` takes a [logical plan](#) and transforms each expression (for every logical operator found in the query plan) as follows:

- For [UnresolvedAttributes](#) with `names` as `groupingid`, `apply` creates a [Alias](#) (with a `GroupingID` child expression and `groupingid name`).

That case seems mostly for compatibility with Hive as `grouping_id` attribute name is used by Hive.

- For [UnresolvedGenerators](#), `apply` requests the [SessionCatalog](#) to [find a Generator function by name](#).

If some other non-generator function is found for the name, `apply` fails the analysis phase by reporting an `AnalysisException` :

```
[name] is expected to be a generator. However, its class is [className], which is not a generator.
```

- For [UnresolvedFunctions](#), `apply` requests the [SessionCatalog](#) to [find a function by name](#).
- [AggregateWindowFunctions](#) are returned directly or `apply` fails the analysis phase by reporting an `AnalysisException` when the `UnresolvedFunction` has `isDistinct` flag enabled.

```
[name] does not support the modifier DISTINCT
```

- [AggregateFunctions](#) are wrapped in a [AggregateExpression](#) (with `complete` aggregate mode)
- All other functions are returned directly or `apply` fails the analysis phase by reporting an `AnalysisException` when the `UnresolvedFunction` has `isDistinct` flag enabled.

```
[name] does not support the modifier DISTINCT
```

`apply` [skips unresolved expressions](#).

ResolveHiveSerdeTable Logical Resolution Rule

`ResolveHiveSerdeTable` is a logical resolution rule (i.e. `Rule[LogicalPlan]`) that the [Hive-specific logical query plan analyzer](#) uses to [resolve the metadata of a hive table](#) for [CreateTable](#) logical operators.

`ResolveHiveSerdeTable` is part of [additional rules](#) in [Resolution](#) fixed-point batch of rules.

```
// FIXME Example of ResolveHiveSerdeTable
```

Applying `ResolveHiveSerdeTable` Rule to Logical Plan — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

`apply ...FIXME`

ResolveInlineTables Logical Resolution Rule

`ResolveInlineTables` is a logical resolution rule that replaces (replaces) `UnresolvedInlineTable` operators to `LocalRelations` in a logical query plan.

`ResolveInlineTables` is part of the `Resolution` fixed-point batch in the standard batches of the `Analyzer`.

`ResolveInlineTables` is simply a `Catalyst` rule for transforming `logical plans`, i.e. `Rule[LogicalPlan]`.

`ResolveInlineTables` takes a `SQLConf` when created.

```
val q = sql("VALUES 1, 2, 3")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'UnresolvedInlineTable [col1], [List(1), List(2), List(3)]

scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.conf
org.apache.spark.sql.internal.SQLConf

import org.apache.spark.sql.catalyst.analysis.ResolveInlineTables
val rule = ResolveInlineTables(spark.sessionState.conf)

val planAfterResolveInlineTables = rule(plan)
scala> println(planAfterResolveInlineTables.numberedTreeString)
00 LocalRelation [col1#2]
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the `Rule Contract` to execute (apply) a rule on a `TreeNode` (e.g. `LogicalPlan`).

`apply` simply searches the input plan up to find `UnresolvedInlineTable` logical operators with `rows` expressions resolved.

For such a `UnresolvedInlineTable` logical operator, `apply validateInputDimension` and `validateInputEvaluable`.

In the end, `apply` converts the `UnresolvedInlineTable` to a `LocalRelation`.

validateInputDimension Internal Method

```
validateInputDimension(table: UnresolvedInlineTable): Unit
```

```
validateInputDimension ...FIXME
```

Note	<code>validateInputDimension</code> is used exclusively when <code>ResolveInlineTables</code> logical resolution rule is executed .
------	---

validateInputEvaluable Internal Method

```
validateInputEvaluable(table: UnresolvedInlineTable): Unit
```

```
validateInputEvaluable ...FIXME
```

Note	<code>validateInputEvaluable</code> is used exclusively when <code>ResolveInlineTables</code> logical resolution rule is executed .
------	---

Converting UnresolvedInlineTable to LocalRelation — convert Internal Method

```
convert(table: UnresolvedInlineTable): LocalRelation
```

```
convert ...FIXME
```

Note	<code>convert</code> is used exclusively when <code>ResolveInlineTables</code> logical resolution rule is executed .
------	--

ResolveMissingReferences

ResolveMissingReferences is...FIXME

resolveExprsAndAddMissingAttrs Internal Method

```
resolveExprsAndAddMissingAttrs(  
    exprs: Seq[Expression],  
    plan: LogicalPlan): (Seq[Expression], LogicalPlan)
```

resolveExprsAndAddMissingAttrs ...FIXME

Note

resolveExprsAndAddMissingAttrs is used when...FIXME

ResolveOrdinalInOrderByAndGroupBy Logical Resolution Rule

`ResolveOrdinalInOrderByAndGroupBy` is a logical resolution rule that converts ordinal positions in Sort and Aggregate logical operators with corresponding expressions in a logical query plan.

`ResolveOrdinalInOrderByAndGroupBy` is part of the [Resolution](#) fixed-point batch in the standard batches of the [Analyzer](#).

`ResolveOrdinalInOrderByAndGroupBy` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`ResolveOrdinalInOrderByAndGroupBy` takes no arguments when created.

```
// FIXME: DEMO
val rule = spark.sessionState.analyzer.ResolveOrdinalInOrderByAndGroupBy

val plan = ???
val planResolved = rule(plan)
scala> println(planResolved.numberedTreeString)
00 'UnresolvedRelation `t1`
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (<code>apply</code>) a rule on a TreeNode (e.g. LogicalPlan).
------	---

`apply` walks the logical plan from children up the tree and looks for Sort and Aggregate logical operators with [UnresolvedOrdinal](#) leaf expressions (in ordering and grouping expressions, respectively).

For a [Sort](#) logical operator with [UnresolvedOrdinal](#) expressions, `apply` replaces all the [SortOrder](#) expressions (with [UnresolvedOrdinal](#) child expressions) with [SortOrder](#) expressions and the expression at the `index - 1` position in the [output schema](#) of the child logical operator.

For a [Aggregate](#) logical operator with [UnresolvedOrdinal](#) expressions, `apply` replaces all the expressions (with [UnresolvedOrdinal](#) child expressions) with the expression at the `index - 1` position in the [aggregate named expressions](#) of the current [Aggregate](#) logical operator.

`apply` throws a [AnalysisException](#) (and hence fails an analysis) if the ordinal is outside the range:

```
ORDER BY position [index] is not in select list (valid range is [1, [output.size]])
GROUP BY position [index] is not in select list (valid range is [1, [aggs.size]])
```

ResolveOutputRelation Logical Resolution Rule

ResolveOutputRelation is...FIXME

ResolveReferences Logical Resolution Rule

`ResolveReferences` is a logical resolution rule that the [logical query plan analyzer](#) uses to [resolve FIXME](#) in a logical query plan, i.e.

1. Resolves...FIXME

Technically, `ResolveReferences` is just a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`ResolveReferences` is part of [Resolution](#) fixed-point batch of rules.

```
// FIXME: Use Catalyst DSL to create a logical plan
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")

import org.apache.spark.sql.catalyst.dsl.expressions._
val logicalPlan = t1
  .select("a".attr, star())
  .groupBy(groupingExprs = "b".attr)(aggregateExprs = star())
scala> println(logicalPlan.numberedTreeString)
00 'Aggregate ['b], [*]
01 +- 'Project ['a, *]
02   +- 'UnresolvedRelation `t1`
// END FIXME

// Make the example repeatable
val table = "t1"
import org.apache.spark.sql.catalyst.TableIdentifier
spark.sessionState.catalog.dropTable(TableIdentifier(table), ignoreIfNotExists = true,
  purge = true)

Seq((0, "zero"), (1, "one")).toDF("id", "name").createOrReplaceTempView(table)
val query = spark.table("t1").select("id", "*").groupBy("id", "name").agg(col("*"))
val logicalPlan = query.queryExecution.logical
scala> println(logicalPlan.numberedTreeString)
00 'Aggregate [id#28, name#29], [id#28, name#29, *]
01 +- Project [id#28, id#28, name#29]
02   +- SubqueryAlias t1
03     +- Project [_1#25 AS id#28, _2#26 AS name#29]
04       +- LocalRelation [_1#25, _2#26]

import spark.sessionState.analyzer.ResolveReferences
val planWithRefsResolved = ResolveReferences(logicalPlan)
scala> println(planWithRefsResolved.numberedTreeString)
00 Aggregate [id#28, name#29], [id#28, name#29, id#28, id#28, name#29]
01 +- Project [id#28, id#28, name#29]
02   +- SubqueryAlias t1
03     +- Project [_1#25 AS id#28, _2#26 AS name#29]
04       +- LocalRelation [_1#25, _2#26]
```

Resolving Expressions of Logical Plan — `resolve` Internal Method

```
resolve(e: Expression, q: LogicalPlan): Expression
```

`resolve` resolves the input expression per type:

1. `UnresolvedAttribute` expressions

2. `UnresolvedExtractValue` expressions
3. All other expressions

Note

`resolve` is used exclusively when `ResolveReferences` is requested to `resolve` reference expressions in a logical query plan.

Resolving Reference Expressions In Logical Query Plan (Applying `ResolveReferences` to Logical Plan) — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

`apply` resolves the following logical operators:

- [Project](#) logical operator with a `star` expression to...FIXME
- [Aggregate](#) logical operator with a `star` expression to...FIXME
- [ScriptTransformation](#) logical operator with a `star` expression to...FIXME
- [Generate](#) logical operator with a `star` expression to...FIXME
- [Join](#) logical operator with `duplicateResolved` ...FIXME
- [Intersect](#) logical operator with `duplicateResolved` ...FIXME
- [Except](#) logical operator with `duplicateResolved` ...FIXME
- [Sort](#) logical operator unresolved with child operators resolved...FIXME
- [Generate](#) logical operator resolved...FIXME
- [Generate](#) logical operator unresolved...FIXME

In the end, `apply` [resolves](#) the expressions of the input logical operator.

`apply` skips logical operators that:

- Use `UnresolvedDeserializer` expressions
- Have child operators [unresolved](#)

Expanding Star Expressions — `buildExpandedProjectList` Internal Method

```
buildExpandedProjectList(  
    exprs: Seq[NamedExpression],  
    child: LogicalPlan): Seq[NamedExpression]
```

`buildExpandedProjectList` expands (*converts*) **Star** expressions in the input **named expressions** recursively (down the expression tree) per expression:

- For a `star` expression, `buildExpandedProjectList` requests it to **expand** given the input `child` logical plan
- For a `unresolvedAlias` expression with a `star` child expression, `buildExpandedProjectList` requests it to **expand** given the input `child` logical plan (similarly to a `star` expression alone in the above case)
- For `exprs` with `star` expressions down the expression tree, `buildExpandedProjectList` **expandStarExpression** passing the input `exprs` and `child`

Note

`buildExpandedProjectList` is used when `ResolveReferences` is requested to **resolve reference expressions** (in `Project` and `Aggregate` operators with `star` expressions).

expandStarExpression Method

```
expandStarExpression(expr: Expression, child: LogicalPlan): Expression
```

`expandStarExpression` expands (*transforms*) the following expressions in the input `expr` expression:

1. For **UnresolvedFunction** expressions with **Star** child expressions, `expandStarExpression` requests the `star` expressions to **expand** given the input `child` logical plan and the **resolver**.

```
// Using Catalyst DSL to create a logical plan with a function with Star child expression
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")

import org.apache.spark.sql.catalyst.dsl.expressions._
val f1 = 'f1.function(star())

val plan = t1.select(f1)
scala> println(plan.numberedTreeString)
00 'Project [unresolvedalias('f1(*), None)]
01 +- 'UnresolvedRelation `t1`

// CAUTION: FIXME How to demo that the plan gets resolved using ResolveReferences.
expandStarExpression?
```

- For [CreateNamedStruct](#) expressions with [Star](#) child expressions among the values, `expandStarExpression` ...FIXME
- For [CreateArray](#) expressions with [Star](#) child expressions, `expandStarExpression` ...FIXME
- For [Murmur3Hash](#) expressions with [Star](#) child expressions, `expandStarExpression` ...FIXME

For any other uses of [Star](#) expressions, `expandStarExpression` fails analysis with a `AnalysisException`:

```
Invalid usage of '*' in expression '[exprName]'
```

Note	<code>expandStarExpression</code> is used exclusively when <code>ResolveReferences</code> is requested to expand Star expressions (in <code>Project</code> and <code>Aggregate</code> operators).
------	---

dedupRight Internal Method

```
dedupRight(left: LogicalPlan, right: LogicalPlan): LogicalPlan
```

`dedupRight` ...FIXME

Note	<code>dedupRight</code> is used when...FIXME
------	--

dedupOuterReferencesInSubquery Internal Method

```
dedupOuterReferencesInSubquery(  
    plan: LogicalPlan,  
    attrMap: AttributeMap[Attribute]): LogicalPlan
```

dedupOuterReferencesInSubquery ...FIXME

Note

dedupOuterReferencesInSubquery is used when...FIXME

ResolveRelations Logical Resolution Rule — Resolving UnresolvedRelations With Tables in Catalog

`ResolveRelations` is a logical resolution rule that the [logical query plan analyzer](#) uses to [resolve UnresolvedRelations](#) (in a logical query plan), i.e.

- Resolves [UnresolvedRelation](#) logical operators (in [InsertIntoTable](#) operators)
- Other uses of `UnresolvedRelation`

Technically, `ResolveRelations` is just a [Catalyst rule](#) for transforming [logical plans](#), i.e.

```
Rule[LogicalPlan] .
```

`ResolveRelations` is part of [Resolution](#) fixed-point batch of rules.

```
// Example: InsertIntoTable with UnresolvedRelation
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = table("t1").insertInto(tableName = "t2", overwrite = true)
scala> println(plan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- 'UnresolvedRelation `t1`

// Register the tables so the following resolution works
sql("CREATE TABLE IF NOT EXISTS t1(id long)")
sql("CREATE TABLE IF NOT EXISTS t2(id long)")

// ResolveRelations is a Scala object of the Analyzer class
// We need an instance of the Analyzer class to access it
import spark.sessionState.analyzer.ResolveRelations
val resolvedPlan = ResolveRelations(plan)
scala> println(resolvedPlan.numberedTreeString)
00 'InsertIntoTable 'UnresolvedRelation `t2`, true, false
01 +- 'SubqueryAlias t1
02     +- 'UnresolvedCatalogRelation `default`.`t1`, org.apache.hadoop.hive.io.parquet.serde.ParquetHiveSerDe

// Example: Other uses of UnresolvedRelation
// Use a temporary view
val v1 = spark.range(1).createOrReplaceTempView("v1")
scala> spark.catalog.listTables.filter($"name" === "v1").show
+-----+-----+-----+
|name|database|description|tableType|isTemporary|
+-----+-----+-----+
| v1|    null|      null|TEMPORARY|     true|
+-----+-----+-----+
```

```

import org.apache.spark.sql.catalyst.dsl.expressions._
val plan = table("v1").select(star())
scala> println(plan.numberedTreeString)
00 'Project [*]
01 +- 'UnresolvedRelation `v1`

val resolvedPlan = ResolveRelations(plan)
scala> println(resolvedPlan.numberedTreeString)
00 'Project [*]
01 +- SubqueryAlias v1
02   +- Range (0, 1, step=1, splits=Some(8))

// Example
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = table(db = "db1", ref = "t1")
scala> println(plan.numberedTreeString)
00 'UnresolvedRelation `db1`.`t1`

// Register the database so the following resolution works
sql("CREATE DATABASE IF NOT EXISTS db1")

val resolvedPlan = ResolveRelations(plan)
scala> println(resolvedPlan.numberedTreeString)
00 'SubqueryAlias t1
01 +- 'UnresolvedCatalogRelation `db1`.`t1`, org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe

```

Applying ResolveRelations to Logical Plan — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	apply is part of Rule Contract to apply a rule to a logical plan.
------	---

apply ...FIXME

Resolving Relation — resolveRelation Method

```
resolveRelation(plan: LogicalPlan): LogicalPlan
```

resolveRelation ...FIXME

Note	resolveRelation is used when...FIXME
------	--------------------------------------

isRunningDirectlyOnFiles Internal Method

```
isRunningDirectlyOnFiles(table: TableIdentifier): Boolean
```

`isRunningDirectlyOnFiles` is enabled (i.e. `true`) when all of the following conditions hold:

- The database of the input `table` is defined
- `spark.sql.runSQLOnFiles` internal configuration property is enabled
- The `table` is not a [temporary table](#)
- The `database` or the `table` do not exist (in the [SessionCatalog](#))

Note

`isRunningDirectlyOnFiles` is used exclusively when `ResolveRelations` [resolves a relation](#) (as a [UnresolvedRelation](#) leaf logical operator for a table reference).

Finding Table in Session-Spaced Catalog of Relational Entities — `lookupTableFromCatalog` Internal Method

```
lookupTableFromCatalog(  
    u: UnresolvedRelation,  
    defaultDatabase: Option[String] = None): LogicalPlan
```

`lookupTableFromCatalog` simply requests `SessionCatalog` to [find the table in relational catalogs](#).

Note

`lookupTableFromCatalog` requests `Analyzer` for the current `SessionCatalog`.

Note

The table is described using `TableIdentifier` of the input `UnresolvedRelation`.

`lookupTableFromCatalog` fails the analysis phase (by reporting a `AnalysisException`) when the table or the table's database cannot be found.

Note

`lookupTableFromCatalog` is used when `ResolveRelations` is executed (for `InsertIntoTable` with `UnresolvedRelation` operators) or [resolves a relation](#) (for "standalone" `UnresolvedRelations`).

ResolveSQLOnFile Logical Evaluation Rule for...FIXME

ResolveSQLOnFile is...FIXME

maybeSQLFile Internal Method

```
maybeSQLFile(u: UnresolvedRelation): Boolean
```

maybeSQLFile is enabled (i.e. true) where the following all hold:

1. FIXME

Note

maybeSQLFile is used exclusively when...FIXME

Applying Rule to Logical Plan — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

apply is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

apply ...FIXME

ResolveSubquery Logical Resolution Rule

`ResolveSubquery` is a **logical resolution** that resolves subquery expressions (`ScalarSubquery`, `Exists` and `In`) when transforming a logical plan with the following logical operators:

1. `Filter` operators with an `Aggregate` child operator
2. Unary operators with the children resolved

`ResolveSubquery` is part of `Resolution` fixed-point batch of rules of the `Spark Analyzer`.

Technically, `ResolveSubquery` is a `Catalyst rule` for transforming `logical plans`, i.e. `Rule[LogicalPlan]`.

```
// Use Catalyst DSL
import org.apache.spark.sql.catalyst.expressions._
val a = 'a.int

import org.apache.spark.sql.catalyst.plans.logical.LocalRelation
val rel = LocalRelation(a)

import org.apache.spark.sql.catalyst.expressions.Literal
val list = Seq[Literal](1)

// FIXME Use a correct query to demo ResolveSubquery
import org.apache.spark.sql.catalyst.plans.logical.Filter
import org.apache.spark.sql.catalyst.expressions.In
val plan = Filter(condition = In(value = a, list), child = rel)

scala> println(plan.numberedTreeString)
00 Filter a#9 IN (1)
01 +- LocalRelation <empty>, [a#9]

import spark.sessionState.analyzer.ResolveSubquery
val analyzedPlan = ResolveSubquery(plan)
scala> println(analyzedPlan.numberedTreeString)
00 Filter a#9 IN (1)
01 +- LocalRelation <empty>, [a#9]
```

Resolving Subquery Expressions (`ScalarSubquery`, `Exists` and `In`) — `resolveSubQueries` Internal Method

```
resolveSubQueries(plan: LogicalPlan, plans: Seq[LogicalPlan]): LogicalPlan
```

`resolveSubQueries` requests the input logical plan to transform expressions (down the operator tree) as follows:

1. For `ScalarSubquery` expressions with `subquery plan` not resolved and `resolveSubQuery` to create resolved `ScalarSubquery` expressions
2. For `Exists` expressions with `subquery plan` not resolved and `resolveSubQuery` to create resolved `Exists` expressions
3. For `In` expressions with `ListQuery` not resolved and `resolveSubQuery` to create resolved `In` expressions

Note

`resolveSubQueries` is used exclusively when `ResolveSubquery` is executed.

resolveSubQuery Internal Method

```
resolveSubQuery(  
    e: SubqueryExpression,  
    plans: Seq[LogicalPlan])(  
    f: (LogicalPlan, Seq[Expression]) => SubqueryExpression): SubqueryExpression
```

`resolveSubQuery ...FIXME`

Note

`resolveSubQuery` is used exclusively when `ResolveSubquery` is requested to resolve subquery expressions (`ScalarSubquery`, `Exists` and `In`).

Applying ResolveSubquery to Logical Plan (Executing ResolveSubquery) — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of `Rule Contract` to apply a rule to a `TreeNode`, e.g. `logical plan`.

`apply` transforms the input `logical plan` as follows:

1. For `Filter` operators with an `Aggregate` operator (as the `child` operator) and the `children resolved`, `apply` resolves subquery expressions (`ScalarSubquery`, `Exists` and `In`) with the `Filter` operator and the plans with the `Aggregate` operator and its single `child`
2. For `unary operators` with the `children resolved`, `apply` resolves subquery expressions (`ScalarSubquery`, `Exists` and `In`) with the unary operator and its single child

ResolveWindowFrame Logical Resolution Rule

`ResolveWindowFrame` is a logical resolution rule that the [logical query plan analyzer](#) uses to validate and resolve [WindowExpression expressions](#) in an entire logical query plan.

Technically, `ResolveWindowFrame` is just a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`ResolveWindowFrame` is part of [Resolution](#) fixed-point batch of rules.

`ResolveWindowFrame` takes a [logical plan](#) and does the following:

1. Makes sure that the window frame of a `WindowFunction` is unspecified or matches the `SpecifiedWindowFrame` of the [WindowSpecDefinition](#) expression.

Reports a `AnalysisException` when the frames do not match:

```
Window Frame [f] must match the required frame [frame]
```

2. Copies the frame specification of `WindowFunction` to [WindowSpecDefinition](#)
3. Creates a new `SpecifiedWindowFrame` for `WindowExpression` with the resolved Catalyst expression and `UnspecifiedFrame`

Note

`ResolveWindowFrame` is a Scala object inside [Analyzer](#) class.

```

import org.apache.spark.sql.expressions.Window
// cume_dist requires ordered windows
val q = spark.
  range(5).
  withColumn("cume_dist", cume_dist() over Window.orderBy("id"))
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
val planBefore: LogicalPlan = q.queryExecution.logical

// Before ResolveWindowFrame
scala> println(planBefore.numberedTreeString)
00 'Project [*], cume_dist() windowspecdefinition('id ASC NULLS FIRST, UnspecifiedFrame
) AS cume_dist#39]
01 +- Range (0, 5, step=1, splits=Some(8))

import spark.sessionState.analyzer.ResolveWindowFrame
val planAfter = ResolveWindowFrame.apply(plan)

// After ResolveWindowFrame
scala> println(planAfter.numberedTreeString)
00 'Project [*], cume_dist() windowspecdefinition('id ASC NULLS FIRST, RANGE BETWEEN UN
BOUNDED PRECEDING AND CURRENT ROW) AS cume_dist#31]
01 +- Range (0, 5, step=1, splits=Some(8))

```

Applying ResolveWindowFrame to Logical Plan — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

`apply ...FIXME`

ResolveWindowOrder Logical Resolution Rule

ResolveWindowOrder is...FIXME

TimeWindowing Logical Resolution Rule

TimeWindowing is a [logical resolution rule](#) that [FIXME](#) in a logical query plan.

TimeWindowing is part of the [Resolution](#) fixed-point batch in the standard batches of the [Analyzer](#).

TimeWindowing is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan] .`

```
// FIXME: DEMO
```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

UpdateOuterReferences Logical Rule

UpdateOuterReferences is...FIXME

Applying UpdateOuterReferences to Logical Plan — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

apply is part of [Rule Contract](#) to apply a rule to a [logical plan](#).

apply ...FIXME

WindowFrameCoercion Type Coercion Logical Rule

`WindowFrameCoercion` is a type coercion logical rule that cast the data types of the boundaries of a range window frame to the data type of the order specification in a `WindowSpecDefinition` in a logical plan.

```

import java.time.LocalDate
import java.sql.Timestamp
val sales = Seq(
  (Timestamp.valueOf(LocalDate.of(2018, 9, 1).atStartOfDay), 5),
  (Timestamp.valueOf(LocalDate.of(2018, 9, 2).atStartOfDay), 10),
  // Mind the 2-day gap
  (Timestamp.valueOf(LocalDate.of(2018, 9, 5).atStartOfDay), 5)
).toDF("time", "volume")
scala> sales.show
+-----+-----+
|       time|volume|
+-----+-----+
|2018-09-01 00:00:00|     5|
|2018-09-02 00:00:00|    10|
|2018-09-05 00:00:00|     5|
+-----+-----+

scala> sales.printSchema
root
 |-- time: timestamp (nullable = true)
 |-- volume: integer (nullable = false)

// FIXME Use Catalyst DSL
// rangeBetween with column expressions
// data type of orderBy expression is date
// data types of range frame boundaries is interval
// WindowSpecDefinition(_, Seq(order), SpecifiedWindowFrame(RangeFrame, lower, upper))
import org.apache.spark.unsafe.types.CalendarInterval
val interval = lit(CalendarInterval.fromString("interval 1 days"))
import org.apache.spark.sql.expressions.Window
val windowSpec = Window.orderBy($"time").rangeBetween(currentRow(), interval)

val q = sales.select(
  $"time",
  (sum($"volume") over windowSpec) as "sum",
  (count($"volume") over windowSpec) as "count")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'Project [unresolvedalias('time, None), sum('volume) windowspecdefinition('time ASC
NULLS FIRST, specifiedwindowframe(RangeFrame, currentrow$, interval 1 days)) AS sum#
156, count('volume) windowspecdefinition('time ASC NULLS FIRST, specifiedwindowframe(R

```

```

RangeFrame, currentRow$, interval 1 days)) AS count#158]
01 +- AnalysisBarrier
02     +- Project [_1#129 AS time#132, _2#130 AS volume#133]
03         +- LocalRelation [_1#129, _2#130]

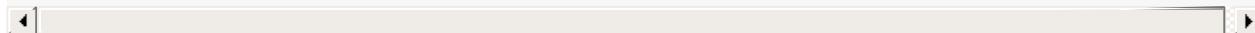
import spark.sessionState.analyzer.ResolveReferences
val planWithRefsResolved = ResolveReferences(plan)

import spark.sessionState.analyzer.ResolveAliases
val planWithAliasesResolved = ResolveReferences(planWithRefsResolved)

// FIXME Looks like nothing changes in the query plan with regard to WindowFrameCoercion

import org.apache.spark.sql.catalyst.analysis.TypeCoercion.WindowFrameCoercion
val afterWindowFrameCoercion = WindowFrameCoercion(planWithRefsResolved)
scala> println(afterWindowFrameCoercion.numberedTreeString)
00 'Project [unresolvedalias(time#132, None), sum(volume#133) windowspecdefinition(time#132 ASC NULLS FIRST, specifiedwindowframe(RangeFrame, currentRow$, interval 1 days)) AS sum#156L, count(volume#133) windowspecdefinition(time#132 ASC NULLS FIRST, specifiedwindowframe(RangeFrame, currentRow$, interval 1 days)) AS count#158L]
01 +- AnalysisBarrier
02     +- Project [_1#129 AS time#132, _2#130 AS volume#133]
03         +- LocalRelation [_1#129, _2#130]

```



```

import java.time.LocalDate
import java.sql.Date
val sales = Seq(
  (Date.valueOf(LocalDate.of(2018, 9, 1)), 5),
  (Date.valueOf(LocalDate.of(2018, 9, 2)), 10),
  // Mind the 2-day gap
  (Date.valueOf(LocalDate.of(2018, 9, 5)), 5)
).toDF("time", "volume")
scala> sales.show
+-----+-----+
|      time|volume|
+-----+-----+
|2018-09-01|     5|
|2018-09-02|    10|
|2018-09-05|     5|
+-----+-----+

scala> sales.printSchema
root
 |-- time: date (nullable = true)
 |-- volume: integer (nullable = false)

// FIXME Use Catalyst DSL
// rangeBetween with column expressions
// data type of orderBy expression is date
// WindowSpecDefinition(_, Seq(order), SpecifiedWindowFrame(RangeFrame, lower, upper))

```

```

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.orderBy($"time").rangeBetween(currentRow(), lit(1))

val q = sales.select(
  $"time",
  (sum($"volume") over windowSpec) as "sum")
val plan = q.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'Project [unresolvedalias('time, None), sum('volume) windowspecdefinition('time ASC
NULLS FIRST, specifiedwindowframe(RangeFrame, currentrow$, 1)) AS sum#238]
01 +- AnalysisBarrier
02     +- Project [_1#222 AS time#225, _2#223 AS volume#226]
03         +- LocalRelation [_1#222, _2#223]

import spark.sessionState.analyzer.ResolveReferences
val planWithRefsResolved = ResolveReferences(plan)

import spark.sessionState.analyzer.ResolveAliases
val planWithAliasesResolved = ResolveReferences(planWithRefsResolved)

// FIXME Looks like nothing changes in the query plan with regard to WindowFrameCoercion

import org.apache.spark.sql.catalyst.analysis.TypeCoercion.WindowFrameCoercion
val afterWindowFrameCoercion = WindowFrameCoercion(planWithAliasesResolved)
scala> println(afterWindowFrameCoercion.numberedTreeString)
00 'Project [unresolvedalias(time#132, None), sum(volume#133) windowspecdefinition(tim
e#132 ASC NULLS FIRST, specifiedwindowframe(RangeFrame, currentrow$, interval 1 days
)) AS sum#156L, count(volume#133) windowspecdefinition(time#132 ASC NULLS FIRST, speci
fiedwindowframe(RangeFrame, currentrow$, interval 1 days)) AS count#158L]
01 +- AnalysisBarrier
02     +- Project [_1#129 AS time#132, _2#130 AS volume#133]
03         +- LocalRelation [_1#129, _2#130]

```

Coercing Types in Logical Plan — `coerceTypes` Method

`coerceTypes(plan: LogicalPlan): LogicalPlan`

Note

`coerceTypes` is part of the [TypeCoercionRule Contract](#) to coerce types in a logical plan.

`coerceTypes` traverses all Catalyst expressions (in the input [LogicalPlan](#)) and replaces the `frameSpecification` of every [WindowSpecDefinition](#) with a `RangeFrame` window frame and the single `order specification` expression resolved with the lower and upper window frame boundary expressions cast to the `data type` of the order specification expression.

createBoundaryCast Internal Method

```
createBoundaryCast(boundary: Expression, dt: DataType): Expression
```

`createBoundaryCast` returns a [Catalyst expression](#) per the input `boundary` [Expression](#) and the `dt` [DataType](#) (in the order of execution):

- The input `boundary` [expression](#) if it is a `SpecialFrameBoundary`
- The input `boundary` [expression](#) if the `dt` [data type](#) is [DateType](#) or [TimestampType](#)
- `Cast` unary operator with the input `boundary` [expression](#) and the `dt` [data type](#) if the [result type](#) of the `boundary` [expression](#) is not the `dt` [data type](#), but the result type can be cast to the `dt` [data type](#)
- The input `boundary` [expression](#)

Note

`createBoundaryCast` is used exclusively when `WindowFrameCoercion` type coercion logical rule is requested to [coerceTypes](#).

WindowsSubstitution Logical Evaluation Rule

`WindowsSubstitution` is a [logical evaluation rule](#) (i.e. `Rule[LogicalPlan]`) that the [logical query plan analyzer](#) uses to resolve (*aka* substitute) `WithWindowDefinition` unary logical operators with `UnresolvedWindowExpression` to their corresponding `WindowExpression` with resolved `WindowSpecDefinition`.

`WindowsSubstitution` is part of [Substitution](#) fixed-point batch of rules.

Note

It appears that `WindowsSubstitution` is exclusively used for pure SQL queries because `WithWindowDefinition` unary logical operator is created exclusively when `AstBuilder` parses window definitions.

If a window specification is not found, `WindowsSubstitution` fails analysis with the following error:

```
Window specification [windowName] is not defined in the WINDOW clause.
```

Note

The analysis failure is unlikely to happen given `AstBuilder` [builds a lookup table of all the named window specifications](#) defined in a SQL text and reports a `ParseException` when a `WindowSpecReference` is not available earlier.

For every `withWindowDefinition`, `WindowsSubstitution` takes the `child` logical plan and transforms its `UnresolvedWindowExpression` expressions to be a `WindowExpression` with a window specification from the `WINDOW` clause (see [WithWindowDefinition Example](#)).

CollapseWindow Logical Optimization

`CollapseWindow` is a base logical optimization that [FIXME](#).

`CollapseWindow` is part of the [Operator Optimization](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`CollapseWindow` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan]` .

```
// FIXME: DEMO
import org.apache.spark.sql.catalyst.optimizer.CollapseWindow

val logicalPlan = ???
val afterCollapseWindow = CollapseWindow(logicalPlan)
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` ...[FIXME](#)

ColumnPruning Logical Optimization

ColumnPruning is a base logical optimization that [FIXME](#).

ColumnPruning is part of the RewriteSubquery once-executed batch in the standard batches of the [Catalyst Optimizer](#).

ColumnPruning is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan]` .

Example 1

```

val dataset = spark.range(10).withColumn("bucket", 'id % 3)

import org.apache.spark.sql.expressions.Window
val rankCol = rank over Window.partitionBy('bucket).orderBy('id) as "rank"

val ranked = dataset.withColumn("rank", rankCol)

scala> ranked.explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ColumnPruning ==
Project [id#73L, bucket#76L, rank#192]
                                         Project
[id#73L, bucket#76L, rank#192]
!+- Project [id#73L, bucket#76L, rank#82, rank#82 AS rank#192]
                                         +- Proj
ect [id#73L, bucket#76L, rank#82 AS rank#192]
    +- Window [rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]      +- W
indow [rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]
!      +- Project [id#73L, bucket#76L]
                                         +
- Project [id#73L, (id#73L % cast(3 as bigint)) AS bucket#76L]
!          +- Project [id#73L, (id#73L % cast(3 as bigint)) AS bucket#76L]

    +- Range (0, 10, step=1, splits=Some(8))
!          +- Range (0, 10, step=1, splits=Some(8))
...
TRACE SparkOptimizer: Fixed point reached for batch Operator Optimizations after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch Operator Optimizations ==
!Project [id#73L, bucket#76L, rank#192]
                                         Window
[rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN UNBOUNDED PREC
EDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]
!+- Project [id#73L, bucket#76L, rank#82, rank#82 AS rank#192]
                                         +- Proj
ect [id#73L, (id#73L % 3) AS bucket#76L]
!    +- Window [rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]      +- R
ange (0, 10, step=1, splits=Some(8))
!      +- Project [id#73L, bucket#76L]
!          +- Project [id#73L, (id#73L % cast(3 as bigint)) AS bucket#76L]
!              +- Range (0, 10, step=1, splits=Some(8))
...

```

Example 2

```

// the business object
case class Person(id: Long, name: String, city: String)

// the dataset to query over
val dataset = Seq(Person(0, "Jacek", "Warsaw")).toDS

// the query
// Note that we work with names only (out of 3 attributes in Person)
val query = dataset.groupBy(upper('name) as 'name).count

scala> query.explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ColumnPruning ==
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
+- LocalRelation [id#125L, name#126, city#127]
+- Project [name#126]
!
+- LocalRelation [id#125L, name#126, city#127]
...
== Parsed Logical Plan ==
'Aggregate [upper('name) AS name#160], [upper('name) AS name#160, count(1) AS count#166
L]
+- LocalRelation [id#125L, name#126, city#127]

== Analyzed Logical Plan ==
name: string, count: bigint
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
+- LocalRelation [id#125L, name#126, city#127]

== Optimized Logical Plan ==
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
+- LocalRelation [name#126]

== Physical Plan ==
*HashAggregate(keys=[upper(name#126)#171], functions=[count(1)], output=[name#160, cou
nt#166L])
+- Exchange hashpartitioning(upper(name#126)#171, 200)
  +- *HashAggregate(keys=[upper(name#126) AS upper(name#126)#171], functions=[partial
  _count(1)], output=[upper(name#126)#171, count#173L])
    +- LocalTableScan [name#126]

```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

CombineTypedFilters Logical Optimization

`CombineTypedFilters` is a [base logical optimization](#) that [combines two back to back \(typed\) filters into one](#) that ultimately ends up as a single method call.

`CombineTypedFilters` is part of the [Object Expressions Optimization](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`CombineTypedFilters` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan]`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

// A query with two consecutive typed filters
val q = spark.range(10).filter(_ % 2 == 0).filter(_ == 0)
scala> q.queryExecution.optimizedPlan
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.CombineTypedFilters ==
  TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], ne
wInstance(class java.lang.Long)      TypedFilter <function1>, class java.lang.Long, [S
tructField(value,LongType,true)], newInstance(class java.lang.Long)
!+- TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], ne
wInstance(class java.lang.Long)  +- Range (0, 10, step=1, splits=Some(8))
!  +- Range (0, 10, step=1, splits=Some(8))

TRACE SparkOptimizer: Fixed point reached for batch Typed Filter Optimization after 2
iterations.
DEBUG SparkOptimizer:
== Result of Batch Typed Filter Optimization ==
  TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], ne
wInstance(class java.lang.Long)      TypedFilter <function1>, class java.lang.Long, [S
tructField(value,LongType,true)], newInstance(class java.lang.Long)
!+- TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], ne
wInstance(class java.lang.Long)  +- Range (0, 10, step=1, splits=Some(8))
!  +- Range (0, 10, step=1, splits=Some(8))
...
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

CombineUnions Logical Optimization

CombineUnions is a base logical optimization that [FIXME](#).

CombineUnions is part of the [Union](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

CombineUnions is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan]` .

```
// FIXME Demo
```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

ComputeCurrentTime Logical Optimization

`ComputeCurrentTime` is a [base logical optimization](#) that [computes the current date and timestamp](#).

`ComputeCurrentTime` is part of the [Finish Analysis](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`ComputeCurrentTime` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

```
// Query with two current_date's
import org.apache.spark.sql.functions.current_date
val q = spark.range(1).select(current_date() as "d1", current_date() as "d2")
val analyzedPlan = q.queryExecution.analyzed

scala> println(analyzedPlan.numberedTreeString)
00 Project [current_date(Some(Europe/Warsaw)) AS d1#12, current_date(Some(Europe/Warsaw))
) AS d2#13]
01 +- Range (0, 1, step=1, splits=Some(8))

import org.apache.spark.sql.catalyst.optimizer.ComputeCurrentTime

val afterComputeCurrentTime = ComputeCurrentTime(analyzedPlan)
scala> println(afterComputeCurrentTime.numberedTreeString)
00 Project [17773 AS d1#12, 17773 AS d2#13]
01 +- Range (0, 1, step=1, splits=Some(8))

// Another query with two current_timestamp's
// Here the millis play a bigger role so it is easier to notice the results
import org.apache.spark.sql.functions.current_timestamp
val q = spark.range(1).select(current_timestamp() as "ts1", current_timestamp() as "ts
2")
val analyzedPlan = q.queryExecution.analyzed
val afterComputeCurrentTime = ComputeCurrentTime(analyzedPlan)
scala> println(afterComputeCurrentTime.numberedTreeString)
00 Project [1535629687768000 AS ts1#18, 1535629687768000 AS ts2#19]
01 +- Range (0, 1, step=1, splits=Some(8))
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

ConstantFolding Logical Optimization

`ConstantFolding` is a base logical optimization that replaces expressions that can be statically evaluated with their equivalent literal values.

`ConstantFolding` is part of the Operator Optimization before Inferring Filters fixed-point batch in the standard batches of the Catalyst Optimizer.

`ConstantFolding` is simply a Catalyst rule for transforming logical plans, i.e.

`Rule[LogicalPlan]`.

```
scala> spark.range(1).select(lit(3) > 2).explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ConstantFolding ==
!Project [(3 > 2) AS (3 > 2)#3]          Project [true AS (3 > 2)#3]
+- Range (0, 1, step=1, splits=Some(8))   +- Range (0, 1, step=1, splits=Some(8))

scala> spark.range(1).select('id + 'id > 0).explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ConstantFolding ==
!Project [((id#7L + id#7L) > cast(0 as bigint)) AS ((id + id) > 0)#10]  Project [((id#
7L + id#7L) > 0) AS ((id + id) > 0)#10]
+- Range (0, 1, step=1, splits=Some(8))                                     +- Range (0, 1
, step=1, splits=Some(8))
```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the Rule Contract to execute (apply) a rule on a `TreeNode` (e.g. `LogicalPlan`).

`apply ...FIXME`

CostBasedJoinReorder Logical Optimization — Join Reordering in Cost-Based Optimization

`CostBasedJoinReorder` is a [base logical optimization](#) that [reorders joins](#) in [cost-based optimization](#).

`ReorderJoin` is part of the [Join Reorder](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`ReorderJoin` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

`CostBasedJoinReorder` [applies](#) the join optimizations on a logical plan with 2 or more [consecutive inner or cross joins](#) (possibly separated by `Project` operators) when `spark.sql.cbo.enabled` and `spark.sql.cbo.joinReorder.enabled` configuration properties are both enabled.

```
// Use shortcuts to read the values of the properties
scala> spark.sessionState.conf.cboEnabled
res0: Boolean = true

scala> spark.sessionState.conf.joinReorderEnabled
res1: Boolean = true
```

`CostBasedJoinReorder` uses [row count](#) statistic that is computed using [ANALYZE TABLE COMPUTE STATISTICS](#) SQL command with no `NOSCAN` option.

```
// Create tables and compute their row count statistics
// There have to be at least 2 joins
// Make the example reproducible
val tableNames = Seq("t1", "t2", "tiny")
import org.apache.spark.sql.catalyst.TableIdentifier
val tableIds = tableNames.map(TableIdentifier.apply)
val sessionCatalog = spark.sessionState.catalog
tableIds.foreach { tableId =>
    sessionCatalog.dropTable(tableId, ignoreIfNotExists = true, purge = true)
}

val belowBroadcastJoinThreshold = spark.sessionState.conf.autoBroadcastJoinThreshold -
1
spark.range(belowBroadcastJoinThreshold).write.saveAsTable("t1")
// t2 is twice as big as t1
spark.range(2 * belowBroadcastJoinThreshold).write.saveAsTable("t2")
spark.range(5).write.saveAsTable("tiny")

// Compute row count statistics
```

```

tableNames.foreach { t =>
    sql(s"ANALYZE TABLE $t COMPUTE STATISTICS")
}

// Load the tables
val t1 = spark.table("t1")
val t2 = spark.table("t2")
val tiny = spark.table("tiny")

// Example: Inner join with join condition
val q = t1.join(t2, Seq("id")).join(tiny, Seq("id"))
val plan = q.queryExecution.analyzed
scala> println(plan.numberedTreeString)
00 Project [id#51L]
01 +- Join Inner, (id#51L = id#57L)
02   :- Project [id#51L]
03     : +- Join Inner, (id#51L = id#54L)
04       :   :- SubqueryAlias t1
05         :     : +- Relation[id#51L] parquet
06       :   +- SubqueryAlias t2
07         :     : +- Relation[id#54L] parquet
08     +- SubqueryAlias tiny
09       : +- Relation[id#57L] parquet

// Eliminate SubqueryAlias logical operators as they no longer needed
// And "confuse" CostBasedJoinReorder
// CostBasedJoinReorder cares about how deep Joins are and reorders consecutive joins
only
import org.apache.spark.sql.catalyst.analysis.EliminateSubqueryAliases
val noAliasesPlan = EliminateSubqueryAliases(plan)
scala> println(noAliasesPlan.numberedTreeString)
00 Project [id#51L]
01 +- Join Inner, (id#51L = id#57L)
02   :- Project [id#51L]
03     : +- Join Inner, (id#51L = id#54L)
04       :   :- Relation[id#51L] parquet
05       :     : +- Relation[id#54L] parquet
06     +- Relation[id#57L] parquet

// Let's go pro and create a custom RuleExecutor (i.e. an Optimizer)
import org.apache.spark.sql.catalyst.rules.RuleExecutor
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.analysis.EliminateSubqueryAliases
import org.apache.spark.sql.catalyst.optimizer.CostBasedJoinReorder
object Optimize extends RuleExecutor[LogicalPlan] {
  val batches =
    Batch("EliminateSubqueryAliases", Once, EliminateSubqueryAliases) :::
    Batch("Join Reorder", Once, CostBasedJoinReorder) :: Nil
}

val joinsReordered = Optimize.execute(plan)
scala> println(joinsReordered.numberedTreeString)
00 Project [id#51L]

```

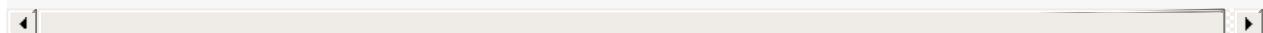
```

01 +- Join Inner, (id#51L = id#54L)
02   :- Project [id#51L]
03   :   +- Join Inner, (id#51L = id#57L)
04   :       :- Relation[id#51L] parquet
05   :       +- Relation[id#57L] parquet
06   +- Relation[id#54L] parquet

// Execute the plans
// Compare the plans as diagrams in web UI @ http://localhost:4040/SQL
// We'd have to use too many internals so let's turn CBO on and off
// Moreover, please remember that the query "phases" are cached
// That's why we copy and paste the entire query for execution
import org.apache.spark.sql.internal.SQLConf
val cc = SQLConf.get
cc.setConf(SQLConf.CBO_ENABLED, false)
val q = t1.join(t2, Seq("id")).join(tiny, Seq("id"))
q.collect.foreach(_ => ())

cc.setConf(SQLConf.CBO_ENABLED, true)
val q = t1.join(t2, Seq("id")).join(tiny, Seq("id"))
q.collect.foreach(_ => ())

```



FIXME Examples of other join queries

Caution

- Cross join with join condition
- Project with attributes only and Inner join with join condition
- Project with attributes only and Cross join with join condition

Enable `DEBUG` logging level for `org.apache.spark.sql.catalyst.optimizer.JoinReorderDP` logger to see the join reordering duration.

Tip

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.catalyst.optimizer.JoinReorderDP=DEBUG
```

Refer to [Logging](#).

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` traverses the input `logical plan` down and tries to `reorder` the following logical operators:

- `Join` for `CROSS` or `INNER` joins with a join condition
- `Project` with the above `Join` child operator and the project list of `Attribute` leaf expressions only

Reordering Logical Plan with Join Operators — `reorder` Internal Method

```
reorder(plan: LogicalPlan, output: Seq[Attribute]): LogicalPlan
```

`reorder` ...FIXME

Note	<code>reorder</code> is used exclusively when <code>CostBasedJoinReorder</code> is applied to a logical plan.
------	---

replaceWithOrderedJoin Internal Method

```
replaceWithOrderedJoin(plan: LogicalPlan): LogicalPlan
```

`replaceWithOrderedJoin` ...FIXME

Note	<code>replaceWithOrderedJoin</code> is used recursively and when <code>CostBasedJoinReorder</code> is <code>reordering</code> ...FIXME
------	--

Extracting Consecutive Join Operators — `extractInnerJoins` Internal Method

```
extractInnerJoins(plan: LogicalPlan): (Seq[LogicalPlan], Set[Expression])
```

`extractInnerJoins` finds consecutive `Join` logical operators (inner or cross) with join conditions or `Project` logical operators with `Join` logical operator and the project list of `Attribute` leaf expressions only.

For `Project` operators `extractInnerJoins` calls itself recursively with the `Join` operator inside.

In the end, `extractInnerJoins` gives the collection of logical plans under the consecutive `Join` logical operators (possibly separated by `Project` operators only) and their join conditions (for which `And` expressions have been split).

Note

`extractInnerJoins` is used recursively when `CostBasedJoinReorder` is reordering a logical plan.

DecimalAggregates Logical Optimization

`DecimalAggregates` is a [base logical optimization](#) that transforms `Sum` and `Average` aggregate functions on fixed-precision `DecimalType` values to use `UnscaledValue` (unscaled `Long`) values in [WindowExpression](#) and [AggregateExpression](#) expressions.

`DecimalAggregates` is part of the [Decimal Optimizations](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`DecimalAggregates` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

Tip Import `DecimalAggregates` and apply the rule directly on your structured queries to see the rule works.

```
import org.apache.spark.sql.catalyst.optimizer.DecimalAggregates
val da = DecimalAggregates(spark.sessionState.conf)

// Build analyzed logical plan
// with sum aggregate function and Decimal field
import org.apache.spark.sql.types.DecimalType
val query = spark.range(5).select(sum($"id" cast DecimalType(1,0)) as "sum")
scala> val plan = query.queryExecution.analyzed
plan: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Aggregate [sum(cast(id#91L as decimal(1,0))) AS sum#95]
+- Range (0, 5, step=1, splits=Some(8))

// Apply DecimalAggregates rule
// Note MakeDecimal and UnscaledValue operators
scala> da.apply(plan)
res27: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Aggregate [MakeDecimal(sum(UnscaledValue(cast(id#91L as decimal(1,0)))),11,0) AS
+- Range (0, 5, step=1, splits=Some(8))
```

Example: sum Aggregate Function on Decimal with Precision Smaller Than 9

```
// sum aggregate with Decimal field with precision <= 8
val q = "SELECT sum(cast(id AS DECIMAL(5,0))) FROM range(1)"

scala> sql(q).explain(true)
== Parsed Logical Plan ==
'Project [unresolvedalias('sum(cast('id as decimal(5,0))), None)]
+- 'UnresolvedTableValuedFunction range, [1]

== Analyzed Logical Plan ==
sum(CAST(id AS DECIMAL(5,0))): decimal(15,0)
Aggregate [sum(cast(id#104L as decimal(5,0))) AS sum(CAST(id AS DECIMAL(5,0)))#106]
+- Range (0, 1, step=1, splits=None)

== Optimized Logical Plan ==
Aggregate [MakeDecimal(sum(UnscaledValue(cast(id#104L as decimal(5,0)))),15,0) AS sum(
CAST(id AS DECIMAL(5,0)))#106]
+- Range (0, 1, step=1, splits=None)

== Physical Plan ==
*HashAggregate(keys=[], functions=[sum(UnscaledValue(cast(id#104L as decimal(5,0))))],
output=[sum(CAST(id AS DECIMAL(5,0)))#106])
+- Exchange SinglePartition
  +- *HashAggregate(keys=[], functions=[partial_sum(UnscaledValue(cast(id#104L as decimal(5,0))))], output=[sum#108L])
    +- *Range (0, 1, step=1, splits=None)
```

Example: avg Aggregate Function on Decimal with Precision Smaller Than 12

```
// avg aggregate with Decimal field with precision <= 11
val q = "SELECT avg(cast(id AS DECIMAL(10,0))) FROM range(1)"

scala> val q = "SELECT avg(cast(id AS DECIMAL(10,0))) FROM range(1)"
q: String = SELECT avg(cast(id AS DECIMAL(10,0))) FROM range(1)

scala> sql(q).explain(true)
== Parsed Logical Plan ==
'Project [unresolvedalias('avg(cast('id as decimal(10,0))), None)]
+- 'UnresolvedTableValuedFunction range, [1]

== Analyzed Logical Plan ==
avg(CAST(id AS DECIMAL(10,0))): decimal(14,4)
Aggregate [avg(cast(id#115L as decimal(10,0))) AS avg(CAST(id AS DECIMAL(10,0)))#117]
+- Range (0, 1, step=1, splits=None)

== Optimized Logical Plan ==
Aggregate [cast((avg(UnscaledValue(cast(id#115L as decimal(10,0)))) / 1.0) as decimal(
14,4)) AS avg(CAST(id AS DECIMAL(10,0)))#117]
+- Range (0, 1, step=1, splits=None)

== Physical Plan ==
*HashAggregate(keys=[], functions=[avg(UnscaledValue(cast(id#115L as decimal(10,0))))])
, output=[avg(CAST(id AS DECIMAL(10,0)))#117])
+- Exchange SinglePartition
    +- *HashAggregate(keys=[], functions=[partial_avg(UnscaledValue(cast(id#115L as decimal(10,0))))], output=[sum#120, count#121L])
        +- *Range (0, 1, step=1, splits=None)
```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

EliminateSerialization Logical Optimization

`EliminateSerialization` is a base logical optimization that optimizes logical plans with `DeserializeToObject` (after `serializeFromObject` or `TypedFilter`), `AppendColumns` (after `SerializeFromObject`), `TypedFilter` (after `serializeFromObject`) logical operators.

`EliminateSerialization` is part of the [Operator Optimization before Inferring Filters](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`EliminateSerialization` is simply a [Catalyst rule](#) for transforming logical plans, i.e. `Rule[LogicalPlan]`.

Examples include:

1. `map` followed by `filter` Logical Plan
2. `map` followed by another `map` Logical Plan
3. `groupByKey` followed by `agg` Logical Plan

Example — `map` followed by `filter` Logical Plan

```

scala> spark.range(4).map(n => n * 2).filter(n => n < 3).explain(extended = true)
== Parsed Logical Plan ==
'TypedFilter <function1>, long, [StructField(value,LongType,false)], unresolveddeserializer(upcast(getcolumnbyordinal(0, LongType), LongType, - root class: "scala.Long"))
+- SerializeFromObject [input[0, bigint, true] AS value#185L]
  +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)]
], obj#184: bigint
  +- DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
    +- Range (0, 4, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: bigint
TypedFilter <function1>, long, [StructField(value,LongType,false)], cast(value#185L as
bigint)
+- SerializeFromObject [input[0, bigint, true] AS value#185L]
  +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)]
], obj#184: bigint
  +- DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
    +- Range (0, 4, step=1, splits=Some(8))

== Optimized Logical Plan ==
SerializeFromObject [input[0, bigint, true] AS value#185L]
+- Filter <function1>.apply
  +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)]
], obj#184: bigint
  +- DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
    +- Range (0, 4, step=1, splits=Some(8))

== Physical Plan ==
*SerializeFromObject [input[0, bigint, true] AS value#185L]
+- *Filter <function1>.apply
  +- *MapElements <function1>, obj#184: bigint
    +- *DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
      +- *Range (0, 4, step=1, splits=Some(8))

```

Example — map followed by another map Logical Plan

```

// Notice unnecessary mapping between String and Int types
val query = spark.range(3).map(_.toString).map(_.toInt)

scala> query.explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.EliminateSerialization ===
SerializeFromObject [input[0, int, true] AS value#91]

```

S

```

    serializeFromObject [input[0, int, true] AS value#91]
      +- MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
      +-
    - MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
    !   +- DeserializeToObject value#86.toString, obj#89: java.lang.String

      +- Project [obj#85 AS obj#89]
    !     +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#86]
          +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
    !     +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
          +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
    !           +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
          +- Range (0, 3, step=1, splits=Some(8))
    !
          +- Range (0, 3, step=1, splits=Some(8))
...
== Parsed Logical Plan ==
'SerializeFromObject [input[0, int, true] AS value#91]
+- 'MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
  +- 'DeserializeToObject unresolveddeserializer(upcast(getcolumnbyordinal(0, StringType), StringType, - root class: "java.lang.String").toString), obj#89: java.lang.String
    +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#86]
      +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
        +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
          +- Range (0, 3, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: int
SerializeFromObject [input[0, int, true] AS value#91]
+- MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
  +- DeserializeToObject cast(value#86 as string).toString, obj#89: java.lang.String
    +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#86]
      +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
        +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
          +- Range (0, 3, step=1, splits=Some(8))

== Optimized Logical Plan ==
SerializeFromObject [input[0, int, true] AS value#91]
+- MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)

```

```

)], obj#90: int
+- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)
], obj#85: java.lang.String
    +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
        +- Range (0, 3, step=1, splits=Some(8))

== Physical Plan ==
*SerializeFromObject [input[0, int, true] AS value#91]
+- *MapElements <function1>, obj#90: int
    +- *MapElements <function1>, obj#85: java.lang.String
        +- *DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Lon
g
    +- *Range (0, 3, step=1, splits=Some(8))

```

Example — groupByKey followed by agg Logical Plan

```

scala> spark.range(4).map(n => (n, n % 2)).groupByKey(_._2).agg(typed.sum(_._2)).expla
in(true)
== Parsed Logical Plan ==
'Aggregate [value#454L], [value#454L, unresolvedalias(typedsumdouble(org.apache.spark.
sql.execution.aggregate.TypedSumDouble@4fcb0de4, Some(unresolveddeserializer(newInstan
ce(class scala.Tuple2), _1#450L, _2#451L)), Some(class scala.Tuple2), Some(StructType(
StructField(_1,LongType,true), StructField(_2,LongType,false))), input[0, double, true]
] AS value#457, unresolveddeserializer(upcast(getcolumnbyordinal(0, DoubleType), Doub
leType, - root class: "scala.Double"), value#457), input[0, double, true] AS value#456,
DoubleType, DoubleType, false), Some(<function1>))]
+- AppendColumns <function1>, class scala.Tuple2, [StructField(_1,LongType,true), Stru
ctField(_2,LongType,false)], newInstance(class scala.Tuple2), [input[0, bigint, true]
AS value#454L]
    +- SerializeFromObject [assertnonnull(input[0, scala.Tuple2, true], top level non-f
lat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true],
top level non-flat input object)._2 AS _2#451L]
        +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,tr
ue)], obj#449: scala.Tuple2
            +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.
Long
                +- Range (0, 4, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: bigint, TypedSumDouble(scala.Tuple2): double
Aggregate [value#454L], [value#454L, typedsumdouble(org.apache.spark.sql.execution.agg
regate.TypedSumDouble@4fcb0de4, Some(newInstance(class scala.Tuple2)), Some(class scal
a.Tuple2), Some(StructType(StructField(_1,LongType,true), StructField(_2,LongType,fals
e))), input[0, double, true] AS value#457, cast(value#457 as double), input[0, double,
true] AS value#456, DoubleType, DoubleType, false) AS TypedSumDouble(scala.Tuple2)#46
2]
+- AppendColumns <function1>, class scala.Tuple2, [StructField(_1,LongType,true), Stru
ctField(_2,LongType,false)], newInstance(class scala.Tuple2), [input[0, bigint, true]
AS value#454L]
    +- SerializeFromObject [assertnonnull(input[0, scala.Tuple2, true], top level non-f
lat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true],

```

```

top level non-flat input object)._2 AS _2#451L]
    +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#449: scala.Tuple2
        +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.Long
Long
    +- Range (0, 4, step=1, splits=Some(8))

== Optimized Logical Plan ==
Aggregate [value#454L], [value#454L, typedsumdouble(org.apache.spark.sql.execution.aggregate.TypedSumDouble@4fc0de4, Some(newInstance(class scala.Tuple2)), Some(class scala.Tuple2), Some(StructType(StructField(_1, LongType, true), StructField(_2, LongType, false))), input[0, double, true] AS value#457, value#457, input[0, double, true] AS value#456, DoubleType, DoubleType, false) AS TypedSumDouble(scala.Tuple2)#462]
+- AppendColumnsWithObject <function1>, [assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._2 AS _2#451L], [input[0, bigint, true] AS value#454L]
    +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#449: scala.Tuple2
        +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.Long
    +- Range (0, 4, step=1, splits=Some(8))

== Physical Plan ==
*HashAggregate(keys=[value#454L], functions=[typedsumdouble(org.apache.spark.sql.execution.aggregate.TypedSumDouble@4fc0de4, Some(newInstance(class scala.Tuple2)), Some(class scala.Tuple2), Some(StructType(StructField(_1, LongType, true), StructField(_2, LongType, false))), input[0, double, true] AS value#457, value#457, input[0, double, true] AS value#456, DoubleType, DoubleType, false)], output=[value#454L, TypedSumDouble(scala.Tuple2)#462])
+- Exchange hashpartitioning(value#454L, 200)
    +- *HashAggregate(keys=[value#454L], functions=[partial_typedsumdouble(org.apache.spark.sql.execution.aggregate.TypedSumDouble@4fc0de4, Some(newInstance(class scala.Tuple2)), Some(class scala.Tuple2), Some(StructType(StructField(_1, LongType, true), StructField(_2, LongType, false))), input[0, double, true] AS value#457, value#457, input[0, double, true] AS value#456, DoubleType, DoubleType, false)], output=[value#454L, value#463])
        +- AppendColumnsWithObject <function1>, [assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._2 AS _2#451L], [input[0, bigint, true] AS value#454L]
            +- MapElements <function1>, obj#449: scala.Tuple2
                +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.Long
                +- *Range (0, 4, step=1, splits=Some(8))

```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

EliminateSubqueryAliases Logical Optimization

`EliminateSubqueryAliases` is a [base logical optimization](#) that removes (eliminates) [SubqueryAlias](#) logical operators from a logical query plan.

`EliminateSubqueryAliases` is part of the [Finish Analysis](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`EliminateSubqueryAliases` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.
`Rule[LogicalPlan]` .

```
// Using Catalyst DSL
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")
val logicalPlan = t1.subquery('a)

import org.apache.spark.sql.catalyst.analysis.EliminateSubqueryAliases
val afterEliminateSubqueryAliases = EliminateSubqueryAliases(logicalPlan)
scala> println(afterEliminateSubqueryAliases.numberedTreeString)
00 'UnresolvedRelation `t1`
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (<code>apply</code>) a rule on a TreeNode (e.g. LogicalPlan).
------	---

`apply` simply removes (eliminates) [SubqueryAlias](#) unary logical operators from the input [logical plan](#).

EliminateView Logical Optimization

`EliminateView` is a base logical optimization that removes (eliminates) View logical operators from a logical query plan.

`EliminateView` is part of the Finish Analysis once-executed batch in the standard batches of the Catalyst Optimizer.

`EliminateView` is simply a Catalyst rule for transforming logical plans, i.e.

`Rule[LogicalPlan]`.

```
val name = "demo_view"
sql(s"CREATE OR REPLACE VIEW $name COMMENT 'demo view' AS VALUES 1,2")
assert(spark.catalog.tableExists(name))

val q = spark.table(name)

val analyzedPlan = q.queryExecution.analyzed
scala> println(analyzedPlan.numberedTreeString)
00 SubqueryAlias demo_view
01 +- View (`default`.`demo_view`, [col1#37])
02   +- Project [cast(col1#38 as int) AS col1#37]
03     +- LocalRelation [col1#38]

import org.apache.spark.sql.catalyst.analysis.EliminateView
val afterEliminateView = EliminateView(analyzedPlan)
// Notice no View operator
scala> println(afterEliminateView.numberedTreeString)
00 SubqueryAlias demo_view
01 +- Project [cast(col1#38 as int) AS col1#37]
02   +- LocalRelation [col1#38]

// TIP: You may also want to use EliminateSubqueryAliases to eliminate SubqueryAliases
```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the Rule Contract to execute (apply) a rule on a TreeNode (e.g. `LogicalPlan`).

`apply` simply removes (eliminates) View unary logical operators from the input logical plan and replaces them with their child logical operator.

apply throws an `AssertionError` when the `output schema` of the `view` operator does not match the `output schema` of the `child` logical operator.

```
assertion failed: The output of the child [output] is different from the view output [output]
```

Note

The assertion should not really happen since `AliasViewChild` logical analysis rule is executed earlier and takes care of not allowing for such difference in the output schema (by throwing an `AnalysisException` earlier).

GetCurrentDatabase Logical Optimization

`GetCurrentDatabase` is a [base logical optimization](#) that [gives the current database](#) for `current_database` SQL function.

`GetCurrentDatabase` is part of the [Finish Analysis](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`GetCurrentDatabase` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

```
val q = sql("SELECT current_database() AS db")
val analyzedPlan = q.queryExecution.analyzed

scala> println(analyzedPlan.numberedTreeString)
00 Project [current_database() AS db#22]
01 +- OneRowRelation

import org.apache.spark.sql.catalyst.optimizer.GetCurrentDatabase

val afterGetCurrentDatabase = GetCurrentDatabase(spark.sessionState.catalog)(analyzedPlan)
scala> println(afterGetCurrentDatabase.numberedTreeString)
00 Project [default AS db#22]
01 +- OneRowRelation
```

Note

`GetCurrentDatabase` corresponds to SQL's `current_database()` function.

You can access the current database in Scala using

```
scala> val database = spark.catalog.currentDatabase
database: String = default
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (`apply`) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` ...FIXME

LimitPushDown Logical Optimization

LimitPushDown is a base logical optimization that transforms the following logical plans:

- LocalLimit with Union
- LocalLimit with Join

LimitPushDown is part of the Operator Optimization before Inferring Filters fixed-point batch in the standard batches of the Catalyst Optimizer.

LimitPushDown is simply a Catalyst rule for transforming logical plans, i.e.

Rule[LogicalPlan] .

```
// test datasets
scala> val ds1 = spark.range(4)
ds1: org.apache.spark.sql.Dataset[Long] = [value: bigint]

scala> val ds2 = spark.range(2)
ds2: org.apache.spark.sql.Dataset[Long] = [value: bigint]

// Case 1. Rather than `LocalLimit` of `Union` do `Union` of `LocalLimit`
scala> ds1.union(ds2).limit(2).explain(true)
== Parsed Logical Plan ==
GlobalLimit 2
+- LocalLimit 2
  +- Union
    :- Range (0, 4, step=1, splits=Some(8))
    +- Range (0, 2, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
GlobalLimit 2
+- LocalLimit 2
  +- Union
    :- Range (0, 4, step=1, splits=Some(8))
    +- Range (0, 2, step=1, splits=Some(8))

== Optimized Logical Plan ==
GlobalLimit 2
+- LocalLimit 2
  +- Union
    :- LocalLimit 2
      :- Range (0, 4, step=1, splits=Some(8))
    +- LocalLimit 2
      :- Range (0, 2, step=1, splits=Some(8))

== Physical Plan ==
CollectLimit 2
+- Union
  :- *LocalLimit 2
    :- *Range (0, 4, step=1, splits=Some(8))
  +- *LocalLimit 2
    :- *Range (0, 2, step=1, splits=Some(8))
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

apply ...FIXME

Creating LimitPushDown Instance

`LimitPushDown` takes the following when created:

- `CatalystConf`

`LimitPushDown` initializes the internal registries and counters.

Note	<code>LimitPushDown</code> is created when
------	--

NullPropagation Logical Optimization — Nullability (NULL Value) Propagation

`NullPropagation` is a [base logical optimization](#) that [FIXME](#).

`NullPropagation` is part of the [Operator Optimization before Inferring Filters](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`NullPropagation` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan]`.

Example: Count Aggregate Operator with Nullable Expressions Only

`NullPropagation` optimization rewrites `Count` aggregate expressions that include expressions that are all nullable to `cast(Literal(0L))`.

```
val table = (0 to 9).toDF("num").as[Int]

// NullPropagation applied
scala> table.select(countDistinct($"num" === null)).explain(true)
== Parsed Logical Plan ==
'Project [count(distinct ('num = null)) AS count(DISTINCT (num = NULL))#45]
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Analyzed Logical Plan ==
count(DISTINCT (num = NULL)): bigint
Aggregate [count(distinct (num#3 = cast(null as int))) AS count(DISTINCT (num = NULL))#45L]
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Optimized Logical Plan ==
Aggregate [0 AS count(DISTINCT (num = NULL))#45L] // <-- HERE
+- LocalRelation

== Physical Plan ==
*HashAggregate(keys=[], functions[], output=[count(DISTINCT (num = NULL))#45L])
+- Exchange SinglePartition
  +- *HashAggregate(keys=[], functions[], output[])
    +- LocalTableScan
```

Example: Count Aggregate Operator with Non-Nullable Non-Distinct Expressions

`NullPropagation` optimization rewrites any non- nullable non-distinct `count` aggregate expressions to `Literal(1)`.

```
val table = (0 to 9).toDF("num").as[Int]

// NullPropagation applied
// current_timestamp() is a non-nullable expression (see the note below)
val query = table.select(count(current_timestamp()) as "count")

scala> println(query.queryExecution.optimizedPlan)
Aggregate [count(1) AS count#64L]
+- LocalRelation

// NullPropagation skipped
val tokens = Seq((0, null), (1, "hello")).toDF("id", "word")
val query = tokens.select(count("word") as "count")

scala> println(query.queryExecution.optimizedPlan)
Aggregate [count(word#55) AS count#71L]
+- LocalRelation [word#55]
```

`count` aggregate expression represents `count` function internally.

Note

```
import org.apache.spark.sql.catalyst.expressions.aggregate.Count
import org.apache.spark.sql.functions.count

scala> count("*").expr.children(0).asInstanceOf[Count]
res0: org.apache.spark.sql.catalyst.expressions.aggregate.Count = count(1)
```

`current_timestamp()` function is non- nullable expression.

Note

```
import org.apache.spark.sql.catalyst.expressions.CurrentTimestamp
import org.apache.spark.sql.functions.current_timestamp

scala> current_timestamp().expr.asInstanceOf[CurrentTimestamp].nullable
res38: Boolean = false
```

Example

```

val table = (0 to 9).toDF("num").as[Int]
val query = table.where('num === null)

scala> query.explain(extended = true)
== Parsed Logical Plan ==
'Filter ('num = null)
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Analyzed Logical Plan ==
num: int
Filter (num#3 = cast(null as int))
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Optimized Logical Plan ==
LocalRelation <empty>, [num#3]

== Physical Plan ==
LocalTableScan <empty>, [num#3]

```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

```
apply ...FIXME
```

OptimizeIn Logical Optimization

`OptimizeIn` is a [base logical optimization](#) that transforms logical plans with `In` predicate expressions as follows:

1. Replaces an `In` expression that has an `empty list` and the `value` expression not `nullable` to `false`
2. Eliminates duplicates of `Literal` expressions in an `In` predicate expression that is `inSetConvertible`
3. Replaces an `In` predicate expression that is `inSetConvertible` with `InSet` expressions when the number of `literal` expressions in the `list` expression is greater than `spark.sql.optimizer.inSetConversionThreshold` internal configuration property (default: `10`)

`OptimizeIn` is part of the [Operator Optimization before Inferring Filters](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`OptimizeIn` is simply a [Catalyst rule](#) for transforming logical plans, i.e. `Rule[LogicalPlan]`.

```
// Use Catalyst DSL to define a logical plan

// HACK: Disable symbolToColumn implicit conversion
// It is imported automatically in spark-shell (and makes demos impossible)
// implicit def symbolToColumn(s: Symbol): org.apache.spark.sql.ColumnName
trait ThatWasABadIdea
implicit def symbolToColumn(ack: ThatWasABadIdea) = ack

import org.apache.spark.sql.catalyst.dsl.expressions._
import org.apache.spark.sql.catalyst.dsl.plans._

import org.apache.spark.sql.catalyst.plans.logical.LocalRelation
val rel = LocalRelation('a.int, 'b.int, 'c.int)

import org.apache.spark.sql.catalyst.expressions.{In, Literal}
val plan = rel
  .where(In('a, Seq[Literal](1, 2, 3)))
  .analyze
scala> println(plan.numberedTreeString)
00 Filter a#6 IN (1,2,3)
01 +- LocalRelation <empty>, [a#6, b#7, c#8]

// In --> InSet
spark.conf.set("spark.sql.optimizer.inSetConversionThreshold", 0)

import org.apache.spark.sql.catalyst.optimizer.OptimizeIn
val optimizedPlan = OptimizeIn(plan)
scala> println(optimizedPlan.numberedTreeString)
00 Filter a#6 INSET (1,2,3)
01 +- LocalRelation <empty>, [a#6, b#7, c#8]
```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

OptimizeSubqueries Logical Optimization

`OptimizeSubqueries` is a [base logical optimization](#) that [FIXME](#).

`OptimizeSubqueries` is part of the [Subquery](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`OptimizeSubqueries` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.
`Rule[LogicalPlan]`.

```
// FIXME Demo
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (apply) a rule on a TreeNode (e.g. LogicalPlan).
------	--

`apply` ...[FIXME](#)

PropagateEmptyRelation Logical Optimization

`PropagateEmptyRelation` is a base logical optimization that collapses plans with empty `LocalRelation` logical operators, e.g. `explode` or `join`.

`PropagateEmptyRelation` is part of the `LocalRelation` fixed-point batch in the standard batches of the `Catalyst Optimizer`.

`PropagateEmptyRelation` is simply a `Catalyst rule` for transforming `logical plans`, i.e. `Rule[LogicalPlan]`.

Explode

```
scala> val emp = spark.emptyDataset[Seq[String]]
emp: org.apache.spark.sql.Dataset[Seq[String]] = [value: array<string>]

scala> emp.select(explode($"value")).show
+---+
|col|
+---+
+---+

scala> emp.select(explode($"value")).explain(true)
== Parsed Logical Plan ==
'Project [explode('value) AS List()]
+- LocalRelation <empty>, [value#77]

== Analyzed Logical Plan ==
col: string
Project [col#89]
+- Generate explode(value#77), false, false, [col#89]
  +- LocalRelation <empty>, [value#77]

== Optimized Logical Plan ==
LocalRelation <empty>, [col#89]

== Physical Plan ==
LocalTableScan <empty>, [col#89]
```

Join

```

scala> spark.emptyDataset[Int].join(spark.range(1)).explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.PropagateEmptyRelation ==
!Join Inner                               LocalRelation <empty>, [value#40, id#42L]
!:- LocalRelation <empty>, [value#40]
!+- Range (0, 1, step=1, splits=Some(8))

TRACE SparkOptimizer: Fixed point reached for batch LocalRelation after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch LocalRelation ==
!Join Inner                               LocalRelation <empty>, [value#40, id#42L]
!:- LocalRelation <empty>, [value#40]
!+- Range (0, 1, step=1, splits=Some(8))
...
== Parsed Logical Plan ==
Join Inner
:- LocalRelation <empty>, [value#40]
+- Range (0, 1, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: int, id: bigint
Join Inner
:- LocalRelation <empty>, [value#40]
+- Range (0, 1, step=1, splits=Some(8))

== Optimized Logical Plan ==
LocalRelation <empty>, [value#40, id#42L]

== Physical Plan ==
LocalTableScan <empty>, [value#40, id#42L]

```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

PullupCorrelatedPredicates Logical Optimization

`PullupCorrelatedPredicates` is a [base logical optimization](#) that transforms logical plans with the following operators:

1. [Filter](#) operators with an [Aggregate](#) child operator
2. [UnaryNode](#) operators

`PullupCorrelatedPredicates` is part of the [Pullup Correlated Expressions](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`PullupCorrelatedPredicates` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

```
Rule[LogicalPlan] .
```

```
import org.apache.spark.sql.catalyst.optimizer.PullupCorrelatedPredicates

// FIXME
// Demo: Filter + Aggregate
// Demo: Filter + UnaryNode

val plan = ???
val optimizedPlan = PullupCorrelatedPredicates(plan)
```

`PullupCorrelatedPredicates` uses [PredicateHelper](#) for...FIXME

pullOutCorrelatedPredicates Internal Method

```
pullOutCorrelatedPredicates(
    sub: LogicalPlan,
    outer: Seq[LogicalPlan]): (LogicalPlan, Seq[Expression])
```

`pullOutCorrelatedPredicates` ...FIXME

Note

`pullOutCorrelatedPredicates` is used exclusively when `PullupCorrelatedPredicates` is requested to [rewriteSubQueries](#).

rewriteSubQueries Internal Method

```
rewriteSubQueries(plan: LogicalPlan, outerPlans: Seq[LogicalPlan]): LogicalPlan
```

```
rewriteSubQueries ...FIXME
```

Note

`rewriteSubQueries` is used exclusively when `PullupCorrelatedPredicates` is [executed](#) (i.e. applied to a [logical plan](#)).

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` transforms the input [logical plan](#) as follows:

1. For [Filter](#) operators with an [Aggregate](#) child operator, `apply` `rewriteSubQueries` with the `Filter` and the `Aggregate` and its `child` as the outer plans
2. For [UnaryNode](#) operators, `apply` `rewriteSubQueries` with the operator and its children as the outer plans

PushDownPredicate — Predicate Pushdown / Filter Pushdown Logical Optimization

`PushDownPredicate` is a [base logical optimization](#) that removes (eliminates) View logical operators from a logical query plan.

`PushDownPredicate` is part of the [Operator Optimization before Inferring Filters](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`PushDownPredicate` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.
`Rule[LogicalPlan]`.

When you execute `where` or `filter` operators right after [loading a dataset](#), Spark SQL will try to push the where/filter predicate down to the data source using a corresponding SQL query with `WHERE` clause (or whatever the proper language for the data source is).

This optimization is called **filter pushdown** or **predicate pushdown** and aims at pushing down the filtering to the "bare metal", i.e. a data source engine. That is to increase the performance of queries since the filtering is performed at the very low level rather than dealing with the entire dataset after it has been loaded to Spark's memory and perhaps causing memory issues.

`PushDownPredicate` is also applied to structured queries with [filters after projections](#) or [filtering on window partitions](#).

Pushing Filter Operator Down Using Projection

```

val dataset = spark.range(2)

scala> dataset.select('id as "_id").filter('_id === 0).explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.PushDownPredicate ==
!Filter (_id#14L = cast(0 as bigint))          Project [id#11L AS _id#14L]
+- Project [id#11L AS _id#14L]                +- Filter (id#11L = cast(0 as bigint))
   +- Range (0, 2, step=1, splits=Some(8))     +- Range (0, 2, step=1, splits=Some(8))
())
...
== Parsed Logical Plan ==
'Filter ('_id = 0)
+- Project [id#11L AS _id#14L]
  +- Range (0, 2, step=1, splits=Some(8))

== Analyzed Logical Plan ==
_id: bigint
Filter (_id#14L = cast(0 as bigint))
+- Project [id#11L AS _id#14L]
  +- Range (0, 2, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#11L AS _id#14L]
+- Filter (id#11L = 0)
  +- Range (0, 2, step=1, splits=Some(8))

== Physical Plan ==
*Project [id#11L AS _id#14L]
+- *Filter (id#11L = 0)
  +- *Range (0, 2, step=1, splits=Some(8))

```

Optimizing Window Aggregate Operators

```

val dataset = spark.range(5).withColumn("group", 'id % 3)
scala> dataset.show
+---+----+
| id|group|
+---+----+
|  0|    0|
|  1|    1|
|  2|    2|
|  3|    0|
|  4|    1|
+---+----+

import org.apache.spark.sql.expressions.Window
val groupW = Window.partitionBy('group).orderBy('id)

// Filter out group 2 after window

```

```

// No need to compute rank for group 2
// Push the filter down
val ranked = dataset.withColumn("rank", rank over groupW).filter('group != 2)

scala> ranked.queryExecution.optimizedPlan
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.PushDownPredicate ==
!Filter NOT (group#35L = cast(2 as bigint))
                                         Proj
ct [id#32L, group#35L, rank#203]
!+- Project [id#32L, group#35L, rank#203]
                                         +- Pr
object [id#32L, group#35L, rank#203, rank#203]
!   +- Project [id#32L, group#35L, rank#203, rank#203]
                                         +- Pr
Window [rank(id#32L) windowspecdefinition(group#35L, id#32L ASC, ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS rank#203], [group#35L], [id#32L ASC]
!     +- Window [rank(id#32L) windowspecdefinition(group#35L, id#32L ASC, ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#203], [group#35L], [id#32L ASC]
+- Project [id#32L, group#35L]
!         +- Project [id#32L, group#35L]
                                         +- Pr
        +- Project [id#32L, (id#32L % cast(3 as bigint)) AS group#35L]
!           +- Project [id#32L, (id#32L % cast(3 as bigint)) AS group#35L]

        +- Filter NOT ((id#32L % cast(3 as bigint)) = cast(2 as bigint))
           +- Range (0, 5, step=1, splits=Some(8))

        +- Range (0, 5, step=1, splits=Some(8))
...
res1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Window [rank(id#32L) windowspecdefinition(group#35L, id#32L ASC, ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS rank#203], [group#35L], [id#32L ASC]
+- Project [id#32L, (id#32L % 3) AS group#35L]
  +- Filter NOT ((id#32L % 3) = 2)
    +- Range (0, 5, step=1, splits=Some(8))

```

JDBC Data Source

Tip

Follow the instructions on how to set up PostgreSQL in [Creating DataFrames from Tables using JDBC and PostgreSQL](#).

Given the following code:

```
// Start with the PostgreSQL driver on CLASSPATH

case class Project(id: Long, name: String, website: String)

// No optimizations for typed queries
// LOG: execute <unnamed>: SELECT "id", "name", "website" FROM projects
val df = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:sparkdb")
  .option("dbtable", "projects")
  .load()
  .as[Project]
  .filter(_.name.contains("Spark"))

// Only the following would end up with the pushdown
val df = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:sparkdb")
  .option("dbtable", "projects")
  .load()
  .where("""name like "%Spark%"""")
```

`PushDownPredicate` translates the above query to the following SQL query:

```
LOG: execute <unnamed>: SELECT "id", "name", "website" FROM projects WHERE (name LIKE '%Spark%')
```

Tip

Enable `all` logs in PostgreSQL to see the above SELECT and other query statements.

```
log_statement = 'all'
```

Add `log_statement = 'all'` to `/usr/local/var/postgres/postgresql.conf` on Mac OS X with PostgreSQL installed using `brew`.

Parquet Data Source

```

val spark: SparkSession = ...
import spark.implicits._

// paste it to REPL individually to make the following line work
case class City(id: Long, name: String)

import org.apache.spark.sql.SaveMode.Overwrite
Seq(
  City(0, "Warsaw"),
  City(1, "Toronto"),
  City(2, "London"),
  City(3, "Redmond"),
  City(4, "Boston")).toDF.write.mode(Overwrite).parquet("cities.parquet")

val cities = spark.read.parquet("cities.parquet").as[City]

// Using DataFrame's Column-based query
scala> cities.where('name === "Warsaw").queryExecution.executedPlan
res21: org.apache.spark.sql.execution.SparkPlan =
*Project [id#128L, name#129]
+- *Filter (isnotnull(name#129) && (name#129 = Warsaw))
   +- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, Input
Paths: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFil
ters: [IsNotNull(name), EqualTo(name,Warsaw)], ReadSchema: struct<id:bigint,name:string
g>

// Using SQL query
scala> cities.where("""name = "Warsaw""").queryExecution.executedPlan
res23: org.apache.spark.sql.execution.SparkPlan =
*Project [id#128L, name#129]
+- *Filter (isnotnull(name#129) && (name#129 = Warsaw))
   +- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, Input
Paths: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFil
ters: [IsNotNull(name), EqualTo(name,Warsaw)], ReadSchema: struct<id:bigint,name:string
g>

// Using Dataset's strongly type-safe filter
// Why does the following not push the filter down?
scala> cities.filter(_.name == "Warsaw").queryExecution.executedPlan
res24: org.apache.spark.sql.execution.SparkPlan =
*Filter <function1>.apply
+- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, InputPat
hs: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFilters
: [], ReadSchema: struct<id:bigint,name:string>

```

Hive Data Source

Caution	FIXME
---------	-------

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply ...FIXME`

PushPredicateThroughJoin Logical Optimization

`PushPredicateThroughJoin` is a [base logical optimization](#) that [FIXME](#).

`PushPredicateThroughJoin` is part of the [Operator Optimization before Inferring Filters](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`PushPredicateThroughJoin` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e.

`Rule[LogicalPlan]`.

```
// FIXME Demo
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` ...[FIXME](#)

ReorderJoin Logical Optimization — Reordering Inner and Cross Joins

`ReorderJoin` is a [logical optimization](#) for [join reordering](#).

`ReorderJoin` applies the join optimizations on a logical plan with 2 or more inner and cross joins with at least one join condition.

Caution	FIXME A diagram of a logical plan tree before and after the rule.
---------	---

Technically, `ReorderJoin` is a [Catalyst rule](#) for transforming [logical plans](#), i.e.

```
Rule[LogicalPlan] .
```

```
// Build analyzed logical plan with at least 3 joins and zero or more filters
val belowBroadcastJoinThreshold = spark.sessionState.conf.autoBroadcastJoinThreshold - 1
val belowBroadcast = spark.range(belowBroadcastJoinThreshold)
val large = spark.range(2 * belowBroadcastJoinThreshold)
val tiny = Seq(1, 2, 3, 4, 5).toDF("id")

val q = belowBroadcast.
  crossJoin(large). // <-- CROSS JOIN of two fairly big datasets
  join(tiny).
  where(belowBroadcast("id") === tiny("id"))
val plan = q.queryExecution.analyzed
scala> println(plan.numberedTreeString)
00 Filter (id#0L = cast(id#9 as bigint))
01 +- Join Inner
02   :- Join Cross
03   :  :- Range (0, 10485759, step=1, splits=Some(8))
04   :  +- Range (0, 20971518, step=1, splits=Some(8))
05   +- Project [value#7 AS id#9]
06     +- LocalRelation [value#7]

// Apply ReorderJoin rule
// ReorderJoin alone is (usually?) not enough
// Let's go pro and create a custom RuleExecutor (i.e. a Optimizer)
import org.apache.spark.sql.catalyst.rules.RuleExecutor
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.analysis.EliminateSubqueryAliases
object Optimize extends RuleExecutor[LogicalPlan] {
  import org.apache.spark.sql.catalyst.optimizer._
  val batches =
    Batch("EliminateSubqueryAliases", Once, EliminateSubqueryAliases) :::
    Batch("Operator Optimization", FixedPoint(maxIterations = 100),
      ConvertToLocalRelation,
      PushDownPredicate,
      PushPredicateThroughJoin) :: Nil
```

```

}

val preOptimizedPlan = Optimize.execute(plan)
// Note Join Cross as a child of Join Inner
scala> println(preOptimizedPlan.numberedTreeString)
00 Join Inner, (id#0L = cast(id#9 as bigint))
01 :- Join Cross
02 : :- Range (0, 10485759, step=1, splits=Some(8))
03 : +- Range (0, 20971518, step=1, splits=Some(8))
04 +- LocalRelation [id#9]

// Time...for...ReorderJoin!
import org.apache.spark.sql.catalyst.optimizer.ReorderJoin
val optimizedPlan = ReorderJoin(preOptimizedPlan)
scala> println(optimizedPlan.numberedTreeString)
00 Join Cross
01 :- Join Inner, (id#0L = cast(id#9 as bigint))
02 : :- Range (0, 10485759, step=1, splits=Some(8))
03 : +- LocalRelation [id#9]
04 +- Range (0, 20971518, step=1, splits=Some(8))

// ReorderJoin works differently when the following holds:
// * starSchemaDetection is enabled
// * cboEnabled is disabled
import org.apache.spark.sql.internal.SQLConf.STARSCHEMA_DETECTION
spark.sessionState.conf.setConf(STARSCHEMA_DETECTION, true)

spark.sessionState.conf.starSchemaDetection
spark.sessionState.conf.cboEnabled

```

`ReorderJoin` is part of the [Operator Optimizations](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

Applying ReorderJoin Rule To Logical Plan (Executing ReorderJoin) — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of Rule Contract to apply a rule to a logical plan .
------	---

`apply` traverses the input [logical plan](#) down and finds the following logical operators for [flattenJoin](#):

- [Filter](#) with a inner or cross [Join](#) child operator
- [Join](#) (of any type)

Note

`apply` uses `ExtractFiltersAndInnerJoins` Scala extractor object (using `unapply` method) to "destructure" a logical plan to its logical operators.

Creating Join Logical Operator (Possibly as Child of Filter Operator) — `createOrderedJoin` Method

```
createOrderedJoin(input: Seq[(LogicalPlan, InnerLike)], conditions: Seq[Expression]): LogicalPlan
```

`createOrderedJoin` takes a collection of pairs of a [logical plan](#) and the [join type](#) with join condition [expressions](#) and...FIXME

Note

`createOrderedJoin` makes sure that the `input` has at least two pairs in the `input`.

Note

`createOrderedJoin` is used recursively when `ReorderJoin` is [applied](#) to a logical plan.

"Two Logical Plans" Case

For two joins exactly (i.e. the `input` has two logical plans and their join types), `createOrderedJoin` partitions (aka *splits*) the input condition expressions to the ones that can be evaluated within a join and not.

`createOrderedJoin` determines the join type of the result join. It chooses [inner](#) if the left and right join types are both inner and [cross](#) otherwise.

`createOrderedJoin` creates a [Join](#) logical operator with the input join conditions combined together using `And` expression and the join type (inner or cross).

If there are condition expressions that [could not be evaluated within a join](#),

`createOrderedJoin` creates a [Filter](#) logical operator with the join conditions combined together using `And` expression and the result join operator as the [child](#) operator.

```

import org.apache.spark.sql.catalyst.expressions.Expression
import org.apache.spark.sql.catalyst.expressions.Literal
val a: Expression = Literal("a")
val b: Expression = Literal("b")
// Use Catalyst DSL to compose expressions
import org.apache.spark.sql.catalyst.dsl.expressions._
val cond1 = a === b

// RowNumber is Unevaluable so it cannot be evaluated within a join
import org.apache.spark.sql.catalyst.expressions.RowNumber
val rn = RowNumber()
import org.apache.spark.sql.catalyst.expressions.Unevaluable
assert(rn.isInstanceOf[Unevaluable])
val cond2 = rn === Literal(2)

val cond3 = Literal.TrueLiteral

// Use Catalyst DSL to create logical plans
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")
val t2 = table("t2")

// Use input with exactly 2 pairs
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.plans.{Cross, Inner, InnerLike}
val input: Seq[(LogicalPlan, InnerLike)] = (t1, Inner) :: (t2, Cross) :: Nil
val conditions: Seq[Expression] = cond1 :: cond2 :: cond3 :: Nil

import org.apache.spark.sql.catalyst.optimizer.ReorderJoin
val plan = ReorderJoin.createOrderedJoin(input, conditions)
scala> println(plan.numberedTreeString)
00 'Filter (row_number() = 2)
01 +- 'Join Cross, ((a = b) && true)
02   :- 'UnresolvedRelation `t1`
03   +- 'UnresolvedRelation `t2`
```

"Three Or More Logical Plans" Case

For three or more [logical plans](#) in the `input`, `createOrderedJoin` takes the first plan and tries to find another that has at least one *matching* join condition, i.e. a logical plan with the following:

1. [Output attributes](#) together with the first plan's output attributes are the superset of the [references](#) of a join condition expression (i.e. both plans are required to resolve join references)

2. References of the join condition **cannot be evaluated** using the first plan's or the current plan's **output attributes** (i.e. neither the first plan nor the current plan themselves are enough to resolve join references)

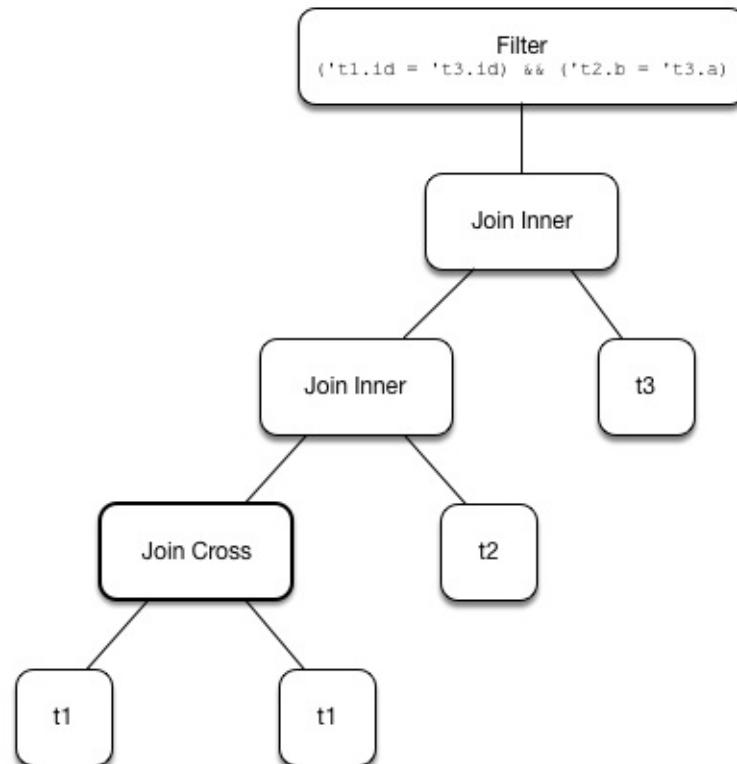


Figure 1. createOrderedJoin with Three Joins (Before)

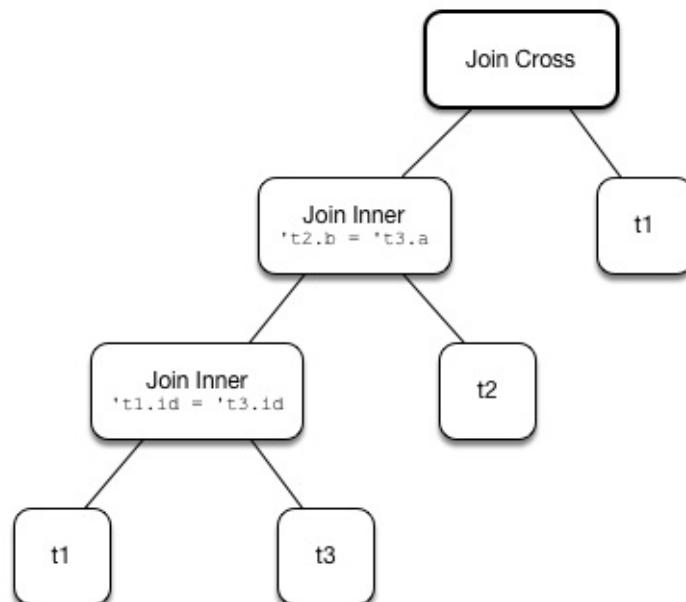


Figure 2. createOrderedJoin with Three Joins (After)

```

// HACK: Disable symbolToColumn implicit conversion
// It is imported automatically in spark-shell (and makes demos impossible)
// implicit def symbolToColumn(s: Symbol): org.apache.spark.sql.ColumnName
trait ThatWasABadIdea
implicit def symbolToColumn(ack: ThatWasABadIdea) = ack
  
```

```

import org.apache.spark.sql.catalyst.plans.logical.LocalRelation
import org.apache.spark.sql.catalyst.dsl.expressions._
import org.apache.spark.sql.catalyst.dsl.plans._
// Note analyze at the end to analyze the queries
val p1 = LocalRelation('id.long, 'a.long, 'b.string).as("t1").where("id".attr != 0).select('id).analyze
val p2 = LocalRelation('id.long, 'b.long).as("t2").analyze
val p3 = LocalRelation('id.long, 'a.string).where("id".attr > 0).select('id, 'id * 2 as "a").as("t3").analyze

// The following input and conditions are equivalent to the following query
val _p1 = Seq((0,1,"one")).toDF("id", "a", "b").as("t1").where(col("id") != 0).select("id")
val _p2 = Seq((0,1)).toDF("id", "b").as("t2")
val _p3 = Seq((0,"one")).toDF("id", "a").where(col("id") > 0).select(col("id"), col("id") * 2 as "a").as("t3")
val _plan = _p1.
  as("p1").
  crossJoin(_p1).
  join(_p2).
  join(_p3).
  where((col("p1.id") === col("t3.id")) && (col("t2.b") === col("t3.a"))).
  queryExecution.
  analyzed
import org.apache.spark.sql.catalyst.planning.ExtractFiltersAndInnerJoins
val Some((plans, conds)) = ExtractFiltersAndInnerJoins.unapply(_plan)

import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.plans.{Cross, Inner, InnerLike}
val input: Seq[(LogicalPlan, InnerLike)] = Seq(
  (p1, Cross),
  (p1, Cross),
  (p2, Inner),
  (p3, Inner))

// (left ++ right).outputSet > expr.references
// ! expr.references > left.outputSet
// ! expr.references > right.outputSet
val p1_id = p1.outputSet.head
val p3_id = p3.outputSet.head
val p2_b = p2.outputSet.tail.head
val p3_a = p3.outputSet.tail.head
val c1 = p1_id === p3_id
val c2 = p2_b === p3_a

// A condition has no references or the references are not a subset of left or right plans
// A couple of assertions that createOrderedJoin does internally
assert(c1.references.nonEmpty)
assert(!c1.references.subsetOf(p1.outputSet))
assert(!c1.references.subsetOf(p3.outputSet))
val refs = p1.analyze.outputSet ++ p3.outputSet
assert(c1.references.subsetOf(refs))

```

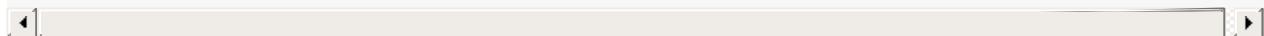
```

import org.apache.spark.sql.catalyst.expressions.Expression
val conditions: Seq[Expression] = Seq(c1, c2)

assert(input.size > 2)
assert(conditions.nonEmpty)

import org.apache.spark.sql.catalyst.optimizer.ReorderJoin
val plan = ReorderJoin.createOrderedJoin(input, conditions)
scala> println(plan.numberedTreeString)
00 'Join Cross
01 :- Join Inner, (b#553L = a#556L)
02 : :- Join Inner, (id#549L = id#554L)
03 : : :- Project [id#549L]
04 : : : +- Filter NOT (id#549L = cast(0 as bigint))
05 : : :     +- LocalRelation <empty>, [id#549L, a#550L, b#551]
06 : : :     +- Project [id#554L, (id#554L * cast(2 as bigint)) AS a#556L]
07 : : :     +- Filter (id#554L > cast(0 as bigint))
08 : : :         +- LocalRelation <empty>, [id#554L, a#555]
09 : : :- LocalRelation <empty>, [id#552L, b#553L]
10 +- Project [id#549L]
11   +- Filter NOT (id#549L = cast(0 as bigint))
12     +- LocalRelation <empty>, [id#549L, a#550L, b#551]

```



`createOrderedJoin` takes the plan that has at least one matching join condition if found or the next plan from the `input` plans.

`createOrderedJoin` partitions (aka *splits*) the input condition expressions to expressions that meet the following requirements (aka *join conditions*) or not (aka *others*):

1. [Expression references](#) being a subset of the [output attributes](#) of the left and the right operators
2. [Can be evaluated within a join](#)

`createOrderedJoin` creates a [Join](#) logical operator with:

1. Left logical operator as the first operator from the `input`
2. Right logical operator as the right as chosen above
3. Join type as the right's join type as chosen above
4. Join conditions combined together using `And` expression

`createOrderedJoin` calls itself recursively with the following:

1. `input` logical joins as a new pair of the new `Join` and `Inner` join type with the remaining logical plans (all but the right)

2. conditions expressions as the *others* conditions (all but the *join conditions* used for the new join)

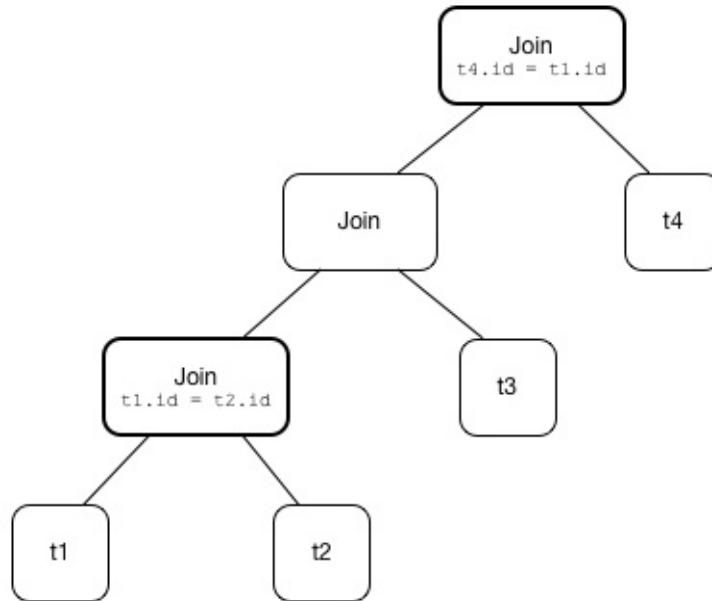


Figure 3. createOrderedJoin with Three Joins

```

import org.apache.spark.sql.catalyst.expressions.Expression
import org.apache.spark.sql.catalyst.expressions.AttributeReference
import org.apache.spark.sql.types.LongType
val t1_id: Expression = AttributeReference(name = "id", LongType)(qualifier = Some("t1"))
val t2_id: Expression = AttributeReference(name = "id", LongType)(qualifier = Some("t2"))
val t4_id: Expression = AttributeReference(name = "id", LongType)(qualifier = Some("t4"))
// Use Catalyst DSL to compose expressions
import org.apache.spark.sql.catalyst.dsl.expressions._
val cond1 = t1_id === t2_id

// RowNumber is Unevaluatable so it cannot be evaluated within a join
import org.apache.spark.sql.catalyst.expressions.RowNumber
val rn = RowNumber()
import org.apache.spark.sql.catalyst.expressions.Unevaluatable
assert(rn.isInstanceOf[Unevaluatable])
import org.apache.spark.sql.catalyst.expressions.Literal
val cond2 = rn === Literal(2)

// That would hardly appear in the condition list
// Just for the demo
val cond3 = Literal.TrueLiteral

val cond4 = t4_id === t1_id

// Use Catalyst DSL to create logical plans
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")
val t2 = table("t2")
  
```

```

val t3 = table("t3")
val t4 = table("t4")

// Use input with 3 or more pairs
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.plans.{Cross, Inner, InnerLike}
val input: Seq[(LogicalPlan, InnerLike)] = Seq(
  (t1, Inner),
  (t2, Inner),
  (t3, Cross),
  (t4, Inner))
val conditions: Seq[Expression] = cond1 :: cond2 :: cond3 :: cond4 :: Nil

import org.apache.spark.sql.catalyst.optimizer.ReorderJoin
val plan = ReorderJoin.createOrderedJoin(input, conditions)
scala> println(plan.numberedTreeString)
00 'Filter (row_number() = 2)
01 +- 'Join Inner, ((id#11L = id#12L) && (id#13L = id#11L))
02   :- 'Join Cross
03     :- 'Join Inner, true
04     : : :- 'UnresolvedRelation `t1`
05     : : +- 'UnresolvedRelation `t2`
06     : +- 'UnresolvedRelation `t3`
07     +- 'UnresolvedRelation `t4`

```

Extracting Filter and Join Operators from Logical Plan — unapply Method (of ExtractFiltersAndInnerJoins)

```
unapply(plan: LogicalPlan): Option[(Seq[(LogicalPlan, InnerLike)], Seq[Expression])]
```

`unapply` extracts `Filter` (with an inner or cross join) or `Join` logical operators (per the input `logical plan`) to...FIXME

Note

`unapply` is a feature of the Scala programming language to define `extractor objects` that take an object and try to give the arguments back. This is most often used in pattern matching and partial functions.

1. For a `Filter` logical operator with a cross or inner `Join` child operator, `unapply flattenJoin` on the `Filter`.
2. For a `Join` logical operator, `unapply flattenJoin` on the `Join`.

```

val d1 = Seq((0, "a"), (1, "b")).toDF("id", "c")
val d2 = Seq((0, "c"), (2, "b")).toDF("id", "c")
val q = d1.join(d2, "id").where($"id" > 0)
val plan = q.queryExecution.analyzed

scala> println(plan.numberedTreeString)
00 Filter (id#34 > 0)
01 +- Project [id#34, c#35, c#44]
02   +- Join Inner, (id#34 = id#43)
03     :- Project [_1#31 AS id#34, _2#32 AS c#35]
04     :   +- LocalRelation [_1#31, _2#32]
05     +- Project [_1#40 AS id#43, _2#41 AS c#44]
06       +- LocalRelation [_1#40, _2#41]

// Let's use Catalyst DSL instead so the plan is cleaner (e.g. no Project in-between)
// We could have used logical rules to clean up the plan
// Leaving the cleaning up as a home exercise for you :)
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")
val t2 = table("t2")
import org.apache.spark.sql.catalyst.expressions.Expression
import org.apache.spark.sql.catalyst.expressions.Literal
val id: Expression = Literal("id")
import org.apache.spark.sql.catalyst.dsl.expressions._
import org.apache.spark.sql.catalyst.plans.Cross
val plan = t1.join(t1, joinType = Cross).join(t2).where(id > 0)
scala> println(plan.numberedTreeString)
00 'Filter (id > 0)
01 +- 'Join Inner
02   :- 'Join Cross
03   :   :- 'UnresolvedRelation `t1`
04   :   +- 'UnresolvedRelation `t1`
05   +- 'UnresolvedRelation `t2`

import org.apache.spark.sql.catalyst.planning.ExtractFiltersAndInnerJoins
// Option[Seq[(LogicalPlan, InnerLike)], Seq[Expression]]
val Some((plans, conditions)) = ExtractFiltersAndInnerJoins.unapply(plan)

assert(plans.size > 2)
assert(conditions.nonEmpty)

CAUTION: FIXME

```

Note

`unapply` is used exclusively when `ReorderJoin` is [executed](#), i.e. applied to a logical plan.

Flattening Consecutive Joins — `flattenJoin` Method (of `ExtractFiltersAndInnerJoins`)

```
flattenJoin(plan: LogicalPlan, parentJoinType: InnerLike = Inner):
  (Seq[(LogicalPlan, InnerLike)], Seq[Expression])
```

`flattenJoin` branches off per the input logical `plan`:

- For an inner or cross `Join` logical operator, `flattenJoin` calls itself recursively with the left-side of the join and the type of the join, and gives:
 1. The logical plans from recursive `flattenJoin` with the right-side of the join and the right join's type
 2. The join conditions from `flattenJoin` with the conditions of the join
- For a `Filter` with an inner or cross `Join` child operator, `flattenJoin` calls itself recursively on the join (that simply removes the `Filter` "layer" and assumes an inner join) and gives:
 1. The logical plans from recursive `flattenJoin`
 2. The join conditions from `flattenJoin` with `Filter`'s conditions
- For all other logical operators, `flattenJoin` gives the input `plan`, the current join type (an inner or cross join) and the empty join condition.

In either case, `flattenJoin` splits *conjunctive predicates*, i.e. removes `And` expressions and gives their child expressions.

```
// Use Catalyst DSL to create a logical plan
// Example 1: One cross join
import org.apache.spark.sql.catalyst.dsl.plans._
val t1 = table("t1")
import org.apache.spark.sql.catalyst.dsl.expressions._
val id = "id".expr
import org.apache.spark.sql.catalyst.plans.cross
val plan = t1.join(t1, joinType = Cross)
scala> println(plan.numberedTreeString)
00 'Join Cross
01 :- 'UnresolvedRelation `t1`
02 +- 'UnresolvedRelation `t1`

import org.apache.spark.sql.catalyst.planning.ExtractFiltersAndInnerJoins
val (plans, conditions) = ExtractFiltersAndInnerJoins.flattenJoin(plan)
assert(plans.size == 2)
assert(conditions.size == 0)

// Example 2: One inner join with a filter
val t2 = table("t2")
val plan = t1.join(t2).where("t1.expr === t2.expr")
scala> println(plan.numberedTreeString)
```

```

00 'Filter (t1 = t2)
01 +- 'Join Inner
02   :- 'UnresolvedRelation `t1`
03   +- 'UnresolvedRelation `t2`

val (plans, conditions) = ExtractFiltersAndInnerJoins.flattenJoin(plan)
assert(plans.size == 2)
assert(conditions.size == 1)

// Example 3: One inner and one cross join with a compound filter
val plan = t1.
  join(t1, joinType = Cross).
  join(t2).
  where("t2.id".expr === "t1.id".expr && "t1.id".expr > 10)
scala> println(plan.numberedTreeString)
00 'Filter ((t2.id = t1.id) && (t1.id > 10))
01 +- 'Join Inner
02   :- 'Join Cross
03   :   :- 'UnresolvedRelation `t1`
04   :   +- 'UnresolvedRelation `t1`
05   +- 'UnresolvedRelation `t2`

val (plans, conditions) = ExtractFiltersAndInnerJoins.flattenJoin(plan)
assert(plans.size == 3)
assert(conditions.size == 2)

// Example 4
val t3 = table("t3")
val plan = t1.
  join(t1, joinType = Cross).
  join(t2).
  where("t2.id".expr === "t1.id".expr && "t1.id".expr > 10).
  join(t3.select(star()))). // <-- just for more fun
  where("t3.id".expr === "t1.id".expr)
scala> println(plan.numberedTreeString)
00 'Filter (t3.id = t1.id)
01 +- 'Join Inner
02   :- 'Filter ((t2.id = t1.id) && (t1.id > 10))
03   :   +- 'Join Inner
04   :     :- 'Join Cross
05   :     :   :- 'UnresolvedRelation `t1`
06   :     :   +- 'UnresolvedRelation `t1`
07   :     +- 'UnresolvedRelation `t2`
08   +- 'Project [*]
09     +- 'UnresolvedRelation `t3`

val (plans, conditions) = ExtractFiltersAndInnerJoins.flattenJoin(plan)
assert(plans.size == 4)
assert(conditions.size == 3)

// Example 5: Join under project is no longer consecutive
val plan = t1.
  join(t1, joinType = Cross).

```

```

select(star()). // <-- separates the cross join from the other joins
join(t2).
where("t2.id".expr === "t1.id".expr && "t1.id".expr > 10).
join(t3.select(star())).
where("t3.id".expr === "t1.id".expr)
scala> println(plan.numberedTreeString)
00 'Filter (t3.id = t1.id)
01 +- 'Join Inner
02   :- 'Filter ((t2.id = t1.id) && (t1.id > 10))
03   :  +- 'Join Inner
04   :    :- 'Project [*]
05   :     :  +- 'Join Cross
06   :      :- 'UnresolvedRelation `t1`
07   :      +- 'UnresolvedRelation `t1`
08   :      +- 'UnresolvedRelation `t2`
09   +- 'Project [*]
10     +- 'UnresolvedRelation `t3`

val (plans, conditions) = ExtractFiltersAndInnerJoins.flattenJoin(plan)
assert(plans.size == 3) // <-- one join less due to Project
assert(conditions.size == 3)

// Example 6: Join on right-hand side is not considered
val plan = t1.
join(
  t1.join(t2).where("t2.id".expr === "t1.id".expr && "t1.id".expr > 10), // <-- join
on RHS
  joinType = Cross).
join(t2).
where("t2.id".expr === "t1.id".expr && "t1.id".expr > 10)
scala> println(plan.numberedTreeString)
00 'Filter ((t2.id = t1.id) && (t1.id > 10))
01 +- 'Join Inner
02   :- 'Join Cross
03   :  :- 'UnresolvedRelation `t1`
04   :  +- 'Filter ((t2.id = t1.id) && (t1.id > 10))
05   :  +- 'Join Inner
06   :    :- 'UnresolvedRelation `t1`
07   :    +- 'UnresolvedRelation `t2`
08   +- 'UnresolvedRelation `t2`

val (plans, conditions) = ExtractFiltersAndInnerJoins.flattenJoin(plan)
assert(plans.size == 3) // <-- one join less due to being on right side
assert(conditions.size == 2)

```

Note

`flattenJoin` is used recursively when `ReorderJoin` is destructures a logical plan (when `executed`).

ReplaceExpressions Logical Optimization

`ReplaceExpressions` is a [base logical optimization](#) that replaces `RuntimeReplaceable` expressions with their single child expression.

`ReplaceExpressions` is part of the [Finish Analysis](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`ReplaceExpressions` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

```
val query = sql("select ifnull(NULL, array('2')) from values 1")
val analyzedPlan = query.queryExecution.analyzed
scala> println(analyzedPlan.numberedTreeString)
00 Project [ifnull(null, array(2)) AS ifnull(NULL, array('2'))#3]
01 +- LocalRelation [col1#2]

import org.apache.spark.sql.catalyst.optimizer.ReplaceExpressions
val optimizedPlan = ReplaceExpressions(analyzedPlan)
scala> println(optimizedPlan.numberedTreeString)
00 Project [coalesce(cast(null as array<string>), cast(array(2) as array<string>)) AS
ifnull(NULL, array('2'))#3]
01 +- LocalRelation [col1#2]
```

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (<code>apply</code>) a rule on a <code>TreeNode</code> (e.g. <code>LogicalPlan</code>).
------	---

`apply` traverses all Catalyst expressions (in the input `LogicalPlan`) and replaces a `RuntimeReplaceable` expression into its single child.

RewriteCorrelatedScalarSubquery Logical Optimization

`RewriteCorrelatedScalarSubquery` is a [base logical optimization](#) that [transforms logical plans](#) with the following operators:

1. `FIXME`

`RewriteCorrelatedScalarSubquery` is part of the [Operator Optimization before Inferring Filters](#) fixed-point batch in the standard batches of the [Catalyst Optimizer](#).

`RewriteCorrelatedScalarSubquery` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

```
import org.apache.spark.sql.catalyst.optimizer.RewriteCorrelatedScalarSubquery

// FIXME
// Demo: Filter + Aggregate
// Demo: Filter + UnaryNode

val plan = ???
val optimizedPlan = RewriteCorrelatedScalarSubquery(plan)
```

evalExpr Internal Method

```
evalExpr(expr: Expression, bindings: Map[ExprId, Option[Any]]) : Option[Any]
```

`evalExpr` ...`FIXME`

Note	<code>evalExpr</code> is used exclusively when <code>RewriteCorrelatedScalarSubquery</code> is... <code>FIXME</code>
------	---

evalAggOnZeroTups Internal Method

```
evalAggOnZeroTups(expr: Expression) : Option[Any]
```

`evalAggOnZeroTups` ...`FIXME`

Note	<code>evalAggOnZeroTups</code> is used exclusively when <code>RewriteCorrelatedScalarSubquery</code> is... <code>FIXME</code>
------	--

evalSubqueryOnZeroTups Internal Method

```
evalSubqueryOnZeroTups(plan: LogicalPlan) : Option[Any]
```

`evalSubqueryOnZeroTups` ...FIXME

Note

`evalSubqueryOnZeroTups` is used exclusively when `RewriteCorrelatedScalarSubquery` is requested to `constructLeftJoins`.

constructLeftJoins Internal Method

```
constructLeftJoins(
    child: LogicalPlan,
    subqueries: ArrayBuffer[ScalarSubquery]): LogicalPlan
```

`constructLeftJoins` ...FIXME

Note

`constructLeftJoins` is used exclusively when `RewriteCorrelatedScalarSubquery` logical optimization is `executed` (i.e. applied to `Aggregate`, `Project` or `Filter` logical operators with correlated scalar subqueries)

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the `Rule Contract` to execute (apply) a rule on a `TreeNode` (e.g. `LogicalPlan`).

`apply` transforms the input `logical plan` as follows:

1. For `Aggregate` operators, `apply` ...FIXME
2. For `Project` operators, `apply` ...FIXME
3. For `Filter` operators, `apply` ...FIXME

Extracting ScalarSubquery Expressions with Children — extractCorrelatedScalarSubqueries Internal Method

```
extractCorrelatedScalarSubqueries[E <: Expression](<br/>  expression: E,<br/>  subqueries: ArrayBuffer[ScalarSubquery]): E
```

`extractCorrelatedScalarSubqueries` finds all `ScalarSubquery` expressions with at least one `child` in the input `expression` and adds them to the input `subqueries` collection.

`extractCorrelatedScalarSubqueries` traverses the input `expression` down (the expression tree) and, every time a `ScalarSubquery` with at least one child is found, returns the head of the output attributes of the `subquery plan`.

In the end, `extractCorrelatedScalarSubqueries` returns the rewritten expression.

Note

`extractCorrelatedScalarSubqueries` uses `scala.collection.mutable.ArrayBuffer` and mutates an instance inside (i.e. adds `ScalarSubquery` expressions) that makes for two output values, i.e. the rewritten expression and the `ScalarSubquery` expressions.

Note

`extractCorrelatedScalarSubqueries` is used exclusively when `RewriteCorrelatedScalarSubquery` is `executed` (i.e. applied to a `logical plan`).

RewritePredicateSubquery Logical Optimization

`RewritePredicateSubquery` is a [base logical optimization](#) that transforms Filter operators with `Exists` and `In` (with `ListQuery`) expressions to Join operators as follows:

- `Filter` operators with `Exists` and `In` with `ListQuery` expressions give **left-semi joins**
- `Filter` operators with `Not` with `Exists` and `In` with `ListQuery` expressions give **left-anti joins**

Note	Prefer <code>EXISTS</code> (over <code>Not</code> with <code>In</code> with <code>ListQuery</code> subquery expression) if performance matters since they say "that will almost certainly be planned as a Broadcast Nested Loop join".
------	--

`RewritePredicateSubquery` is part of the [RewriteSubquery](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

`RewritePredicateSubquery` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

```
// FIXME Examples of RewritePredicateSubquery
// 1. Filters with Exists and In (with ListQuery) expressions
// 2. NOTs

// Based on RewriteSubquerySuite
// FIXME Contribute back to RewriteSubquerySuite
import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
import org.apache.spark.sql.catalyst.rules.RuleExecutor
object Optimize extends RuleExecutor[LogicalPlan] {
    import org.apache.spark.sql.catalyst.optimizer._
    val batches = Seq(
        Batch("Column Pruning", FixedPoint(100), ColumnPruning),
        Batch("Rewrite Subquery", Once,
            RewritePredicateSubquery,
            ColumnPruning,
            CollapseProject,
            RemoveRedundantProject))
}

val q = ...
val optimized = Optimize.execute(q.analyze)
```

`RewritePredicateSubquery` is part of the [RewriteSubquery](#) once-executed batch in the standard batches of the [Catalyst Optimizer](#).

rewriteExistentialExpr Internal Method

```
rewriteExistentialExpr(  
    exprs: Seq[Expression],  
    plan: LogicalPlan): (Option[Expression], LogicalPlan)
```

`rewriteExistentialExpr` ...FIXME

Note

`rewriteExistentialExpr` is used when...FIXME

dedupJoin Internal Method

```
dedupJoin(joinPlan: LogicalPlan): LogicalPlan
```

`dedupJoin` ...FIXME

Note

`dedupJoin` is used when...FIXME

getValueExpression Internal Method

```
getValueExpression(e: Expression): Seq[Expression]
```

`getValueExpression` ...FIXME

Note

`getValueExpression` is used when...FIXME

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply` transforms [Filter](#) unary operators in the input [logical plan](#).

`apply` splits conjunctive predicates in the condition expression (i.e. expressions separated by `And` expression) and then partitions them into two collections of expressions with and without `In` or `Exists` subquery expressions.

`apply` creates a `Filter` operator for condition (sub)expressions without subqueries (combined with `And` expression) if available or takes the `child` operator (of the input `Filter` unary operator).

In the end, `apply` creates a new logical plan with `Join` operators for `Exists` and `In` expressions (and their negations) as follows:

- For `Exists` predicate expressions, `apply rewriteExistentialExpr` and creates a `Join` operator with `LeftSemi` join type. In the end, `apply dedupJoin`
- For `Not` expressions with a `Exists` predicate expression, `apply rewriteExistentialExpr` and creates a `Join` operator with `LeftAnti` join type. In the end, `apply dedupJoin`
- For `In` predicate expressions with a `ListQuery` subquery expression, `apply getValueExpression` followed by `rewriteExistentialExpr` and creates a `Join` operator with `LeftSemi` join type. In the end, `apply dedupJoin`
- For `Not` expressions with a `In` predicate expression with a `ListQuery` subquery expression, `apply getValueExpression`, `rewriteExistentialExpr` followed by `splitting conjunctive predicates` and creates a `Join` operator with `LeftAnti` join type. In the end, `apply dedupJoin`
- For other predicate expressions, `apply rewriteExistentialExpr` and creates a `Project` unary operator with a `Filter` operator

SimplifyCasts Logical Optimization

`SimplifyCasts` is a base logical optimization that eliminates redundant casts in the following cases:

1. The input is already the type to cast to.
2. The input is of `ArrayType` or `MapType` type and contains no `null` elements.

`SimplifyCasts` is part of the Operator Optimization before Inferring Filters fixed-point batch in the standard batches of the Catalyst Optimizer.

`SimplifyCasts` is simply a Catalyst rule for transforming logical plans, i.e.

`Rule[LogicalPlan]`.

```
// Case 1. The input is already the type to cast to
scala> val ds = spark.range(1)
ds: org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> ds.printSchema
root
| -- id: long (nullable = false)

scala> ds.selectExpr("CAST (id AS long)").explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.SimplifyCasts ==
!Project [cast(id#0L as bigint) AS id#7L]   Project [id#0L AS id#7L]
  +- Range (0, 1, step=1, splits=Some(8))    +- Range (0, 1, step=1, splits=Some(8))

TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.RemoveAliasOnlyProject ==
!Project [id#0L AS id#7L]                     Range (0, 1, step=1, splits=Some(8))
  +- Range (0, 1, step=1, splits=Some(8))

TRACE SparkOptimizer: Fixed point reached for batch Operator Optimizations after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch Operator Optimizations ==
!Project [cast(id#0L as bigint) AS id#7L]   Range (0, 1, step=1, splits=Some(8))
  +- Range (0, 1, step=1, splits=Some(8))
...
== Parsed Logical Plan ==
'Project [unresolvedalias(cast('id as bigint), None)]
  +- Range (0, 1, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
```

```

Project [cast(id#0L as bigint) AS id#7L]
+- Range (0, 1, step=1, splits=Some(8))

== Optimized Logical Plan ==
Range (0, 1, step=1, splits=Some(8))

== Physical Plan ==
*Range (0, 1, step=1, splits=Some(8))

// Case 2A. The input is of `ArrayType` type and contains no `null` elements.
scala> val intArray = Seq(Array(1)).toDS
intArray: org.apache.spark.sql.Dataset[Array[Int]] = [value: array<int>]

scala> intArray.printSchema
root
|-- value: array (nullable = true)
|   |-- element: integer (containsNull = false)

scala> intArray.map(arr => arr.sum).explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.SimplifyCasts ===
SerializeFromObject [input[0, int, true] AS value#36]
    SerializeFromObject [input[0, int, true] AS value#36]
    +- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int    +- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int
!    +- DeserializeToObject cast(value#15 as array<int>).toIntArray, obj#34: [I
        +- DeserializeToObject value#15.toIntArray, obj#34: [I
        +- LocalRelation [value#15]
            +- LocalRelation [value#15]

TRACE SparkOptimizer: Fixed point reached for batch Operator Optimizations after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch Operator Optimizations ===
SerializeFromObject [input[0, int, true] AS value#36]
    SerializeFromObject [input[0, int, true] AS value#36]
    +- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int    +- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int
!    +- DeserializeToObject cast(value#15 as array<int>).toIntArray, obj#34: [I
        +- DeserializeToObject value#15.toIntArray, obj#34: [I
        +- LocalRelation [value#15]
            +- LocalRelation [value#15]
    ...
== Parsed Logical Plan ==
'SerializeFromObject [input[0, int, true] AS value#36]
+- 'MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int
    +- 'DeserializeToObject unresolveddeserializer(upcast(getcolumnbyordinal(0, ArrayType(IntegerType, false)), ArrayType(IntegerType, false), - root class: "scala.Array").toIntArray), obj#34: [I

```

```

    +- LocalRelation [value#15]

== Analyzed Logical Plan ==
value: int
SerializeFromObject [input[0, int, true] AS value#36]
+- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType,false),true)], obj#35: int
    +- DeserializeToObject cast(value#15 as array<int>).toIntArray, obj#34: [I
        +- LocalRelation [value#15]

== Optimized Logical Plan ==
SerializeFromObject [input[0, int, true] AS value#36]
+- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType,false),true)], obj#35: int
    +- DeserializeToObject value#15.toIntArray, obj#34: [I
        +- LocalRelation [value#15]

== Physical Plan ==
*SerializeFromObject [input[0, int, true] AS value#36]
+- *MapElements <function1>, obj#35: int
    +- *DeserializeToObject value#15.toIntArray, obj#34: [I
        +- LocalTableScan [value#15]

// Case 2B. The input is of `MapType` type and contains no `null` elements.
scala> val mapDF = Seq(("one", 1), ("two", 2)).toDF("k", "v").withColumn("m", map(col("k"), col("v")))
mapDF: org.apache.spark.sql.DataFrame = [k: string, v: int ... 1 more field]

scala> mapDF.printSchema
root
|-- k: string (nullable = true)
|-- v: integer (nullable = false)
|-- m: map (nullable = false)
|   |-- key: string
|   |-- value: integer (valueContainsNull = false)

scala> mapDF.selectExpr("""CAST (m AS map<string, int>)""").explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.SimplifyCasts ==
!Project [cast(map(_1#250, _2#251) as map<string,int>) AS m#272]   Project [map(_1#250
, _2#251) AS m#272]
    +- LocalRelation [_1#250, _2#251]                                     +- LocalRelation [_
1#250, _2#251]
...
== Parsed Logical Plan ==
'Project [unresolvedalias(cast('m as map<string,int>), None)]
+- Project [k#253, v#254, map(k#253, v#254) AS m#258]
    +- Project [_1#250 AS k#253, _2#251 AS v#254]
        +- LocalRelation [_1#250, _2#251]

== Analyzed Logical Plan ==
m: map<string,int>

```

```
Project [cast(m#258 as map<string,int>) AS m#272]
+- Project [k#253, v#254, map(k#253, v#254) AS m#258]
  +- Project [_1#250 AS k#253, _2#251 AS v#254]
    +- LocalRelation [_1#250, _2#251]

== Optimized Logical Plan ==
LocalRelation [m#272]

== Physical Plan ==
LocalTableScan [m#272]
```

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

```
apply ...FIXME
```

ExtractPythonUDFFromAggregate Logical Optimization

ExtractPythonUDFFromAggregate is...FIXME

apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

apply is part of [Rule Contract](#) to apply a rule to a [TreeNode](#), e.g. [logical query plan](#).

apply ...FIXME

OptimizeMetadataOnlyQuery Logical Optimization

OptimizeMetadataOnlyQuery is...FIXME

apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

apply is part of [Rule Contract](#) to apply a rule to a [TreeNode](#), e.g. [logical query plan](#).

apply ...FIXME

PruneFileSourcePartitions Logical Optimization

`PruneFileSourcePartitions` is...FIXME

apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of [Rule Contract](#) to apply a rule to a [TreeNode](#), e.g. [logical query plan](#).

`apply` ...FIXME

PushDownOperatorsToDataSource Logical Optimization

`PushDownOperatorsToDataSource` is a **logical optimization** that pushes down operators to **underlying data sources** (i.e. `DataSourceV2Relations`) (before planning so that data source can report statistics more accurately).

Technically, `PushDownOperatorsToDataSource` is a [Catalyst rule](#) for transforming logical plans, i.e. `Rule[LogicalPlan]`.

`PushDownOperatorsToDataSource` is part of the [Push down operators to data source scan](#) once-executed rule batch of the [SparkOptimizer](#).

Executing Rule — `apply` Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note	<code>apply</code> is part of the Rule Contract to execute (<code>apply</code>) a rule on a <code>TreeNode</code> (e.g. <code>LogicalPlan</code>).
------	---

`apply` ...FIXME

pushDownRequiredColumns Internal Method

```
pushDownRequiredColumns(plan: LogicalPlan, requiredByParent: AttributeSet): LogicalPlan
```

`pushDownRequiredColumns` branches off per the input [logical operator](#) (that is supposed to have at least one child node):

1. For [Project](#) unary logical operator, `pushDownRequiredColumns` takes the [references](#) of the [project expressions](#) as the required columns (attributes) and executes itself recursively on the [child logical operator](#)

Note that the input `requiredByParent` attributes are not considered in the required columns.

2. For [Filter](#) unary logical operator, `pushDownRequiredColumns` adds the [references](#) of the [filter condition](#) to the input `requiredByParent` attributes and executes itself recursively on the [child logical operator](#)

3. For [DataSourceV2Relation](#) unary logical operator, `pushDownRequiredColumns` ...FIXME
4. For other logical operators, `pushDownRequiredColumns` simply executes itself (using `TreeNode.mapChildren`) recursively on the [child nodes](#) (logical operators)

Note

`pushDownRequiredColumns` is used exclusively when `PushDownOperatorsToDataSource` logical optimization is requested to [execute](#).

Deconstructing Logical Operator — `FilterAndProject.unapply` Method

```
unapply(plan: LogicalPlan): Option[(Seq[NamedExpression], Expression, DataSourceV2Relation)]
```

`unapply` is part of `FilterAndProject` extractor object to destructure the input [logical operator](#) into a tuple with...FIXME

`unapply` works with (matches) the following logical operators:

1. For a [Filter](#) with a [DataSourceV2Relation](#) leaf logical operator, `unapply` ...FIXME
2. For a [Filter](#) with a [Project](#) over a [DataSourceV2Relation](#) leaf logical operator, `unapply` ...FIXME
3. For others, `unapply` returns `None` (i.e. does nothing / does not match)

Note

`unapply` is used exclusively when `PushDownOperatorsToDataSource` logical optimization is requested to [execute](#).

Aggregation Execution Planning Strategy for Aggregate Physical Operators

`Aggregation` is an [execution planning strategy](#) that [SparkPlanner](#) uses to [select aggregate physical operator](#) for [Aggregate](#) logical operator in a [logical query plan](#).

```
scala> :type spark
org.apache.spark.sql.SparkSession

// structured query with count aggregate function
val q = spark
  .range(5)
  .groupBy($"id" % 2 as "group")
  .agg(count("id") as "count")
val plan = q.queryExecution.optimizedPlan
scala> println(plan.numberedTreeString)
00 Aggregate [(id#0L % 2)], [(id#0L % 2) AS group#3L, count(1) AS count#8L]
01 +- Range (0, 5, step=1, splits=Some(8))

import spark.sessionState.planner.Aggregation
val physicalPlan = Aggregation.apply(plan)

// HashAggregateExec selected
scala> println(physicalPlan.head.numberedTreeString)
00 HashAggregate(keys=[(id#0L % 2)#12L], functions=[count(1)], output=[group#3L, count#8L])
01 +- HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#12L], functions=[partial_count(1)],
  output=[(id#0L % 2)#12L, count#14L])
02     +- PlanLater Range (0, 5, step=1, splits=Some(8))
```

`Aggregation` can select the following aggregate physical operators (in the order of preference):

1. [HashAggregateExec](#)
2. [ObjectHashAggregateExec](#)
3. [SortAggregateExec](#)

Applying Aggregation Strategy to Logical Plan (Executing Aggregation) — `apply` Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

Note

`apply` is part of [GenericStrategy Contract](#) to generate a collection of [SparkPlans](#) for a given [logical plan](#).

`apply` requests [PhysicalAggregation](#) extractor for [Aggregate logical operators](#) and creates a single aggregate physical operator for every [Aggregate](#) logical operator found.

Internally, `apply` requests [PhysicalAggregation](#) to [destructure a Aggregate logical operator](#) (into a four-element tuple) and splits [aggregate expressions](#) per whether they are distinct or not (using their [isDistinct](#) flag).

`apply` then creates a physical operator using the following helper methods:

- [AggUtils.planAggregateWithoutDistinct](#) when no distinct aggregate expression is used
- [AggUtils.planAggregateWithOneDistinct](#) when at least one distinct aggregate expression is used.

BasicOperators Execution Planning Strategy

`BasicOperators` is an [execution planning strategy](#) (of `SparkPlanner`) that in general does simple [conversions](#) from [logical operators](#) to their [physical counterparts](#).

Table 1. BasicOperators' Logical to Physical Operator Conversions

Logical Operator	Physical Operator
<code>RunnableCommand</code>	<code>ExecutedCommandExec</code>
<code>MemoryPlan</code>	<code>LocalTableScanExec</code>
<code>DeserializeToObject</code>	<code>DeserializeToObjectExec</code>
<code>SerializeFromObject</code>	<code>SerializeFromObjectExec</code>
<code>MapPartitions</code>	<code>MapPartitionsExec</code>
<code>MapElements</code>	<code>MapElementsExec</code>
<code>AppendColumns</code>	<code>AppendColumnsExec</code>
<code>AppendColumnsWithObject</code>	<code>AppendColumnsWithObjectExec</code>
<code>MapGroups</code>	<code>MapGroupsExec</code>
<code>CoGroup</code>	<code>CoGroupExec</code>
<code>Repartition</code> (with shuffle enabled)	<code>ShuffleExchangeExec</code>
<code>Repartition</code>	<code>CoalesceExec</code>
<code>SortPartitions</code>	<code>SortExec</code>
<code>Sort</code>	<code>SortExec</code>
<code>Project</code>	<code>ProjectExec</code>
<code>Filter</code>	<code>FilterExec</code>
<code>TypedFilter</code>	<code>FilterExec</code>
<code>Expand</code>	<code>ExpandExec</code>
<code>Window</code>	<code>WindowExec</code>

Sample	SampleExec
LocalRelation	LocalTableScanExec
LocalLimit	LocalLimitExec
GlobalLimit	GlobalLimitExec
Union	UnionExec
Generate	GenerateExec
OneRowRelation	RDDScanExec
Range	RangeExec
RepartitionByExpression	ShuffleExchangeExec
ExternalRDD	ExternalRDDScanExec
LogicalRDD	RDDScanExec

Tip	Confirm the operator mapping in the source code of BasicOperators .
-----	---

Note	BasicOperators expects that <code>Distinct</code> , <code>Intersect</code> , and <code>Except</code> logical operators are not used in a logical plan and throws a <code>IllegalStateException</code> if not.
------	---

DataSourceStrategy Execution Planning Strategy

`DataSourceStrategy` is an [execution planning strategy](#) (of [SparkPlanner](#)) that [plans](#) [LogicalRelation](#) logical operators as [RowDataSourceScanExec](#) physical operators (possibly under `FilterExec` and `ProjectExec` operators).

Table 1. `DataSourceStrategy`'s Selection Requirements (in execution order)

Logical Operator	Description
<code>LogicalRelation</code> with a <code>CatalystScan</code> relation	Uses pruneFilterProjectRaw (with the RDD conversion to RDD[InternalRow] as part of <code>scanBuilder</code>). <code>CatalystScan</code> does not seem to be used in Spark SQL.
<code>LogicalRelation</code> with <code>PrunedFilteredScan</code> relation	Uses pruneFilterProject (with the RDD conversion to RDD[InternalRow] as part of <code>scanBuilder</code>). Matches JDBCRelation exclusively
<code>LogicalRelation</code> with a <code>PrunedScan</code> relation	Uses pruneFilterProject (with the RDD conversion to RDD[InternalRow] as part of <code>scanBuilder</code>). <code>PrunedScan</code> does not seem to be used in Spark SQL.
<code>LogicalRelation</code> with a <code>TableScan</code> relation	Creates a RowDataSourceScanExec directly (requesting the <code>TableScan</code> to <code>buildScan</code> followed by RDD conversion to RDD[InternalRow]) Matches KafkaRelation exclusively

```
import org.apache.spark.sql.execution.datasources.DataSourceStrategy
val strategy = DataSourceStrategy(spark.sessionState.conf)

import org.apache.spark.sql.catalyst.plans.logical.LogicalPlan
val plan: LogicalPlan = ???

val sparkPlan = strategy(plan).head
```

Note

`DataSourceStrategy` uses [PhysicalOperation](#) Scala extractor object to destructure a logical query plan.

pruneFilterProject Internal Method

```
pruneFilterProject(
    relation: LogicalRelation,
    projects: Seq[NamedExpression],
    filterPredicates: Seq[Expression],
    scanBuilder: (Seq[Attribute], Array[Filter]) => RDD[InternalRow])
```

`pruneFilterProject` simply calls `pruneFilterProjectRaw` with `scanBuilder` ignoring the `Seq[Expression]` input parameter.

Note

`pruneFilterProject` is used when `DataSourceStrategy` execution planning strategy is [executed](#) (for `LogicalRelation` logical operators with a `PrunedFilteredScan` or a `PrunedScan`).

Selecting Catalyst Expressions Convertible to Data Source Filter Predicates (and Handled by `BaseRelation`) — `selectFilters` Method

```
selectFilters(
    relation: BaseRelation,
    predicates: Seq[Expression]): (Seq[Expression], Seq[Filter], Set[Filter])
```

`selectFilters` builds a map of [Catalyst predicate expressions](#) (from the input `predicates`) that can be [translated](#) to a [data source filter predicate](#).

`selectFilters` then requests the input `BaseRelation` for [unhandled filters](#) (out of the convertible ones that `selectFilters` built the map with).

In the end, `selectFilters` returns a 3-element tuple with the following:

1. Inconvertible and unhandled Catalyst predicate expressions
2. All converted data source filters
3. Pushed-down data source filters (that the input `BaseRelation` can handle)

Note

`selectFilters` is used exclusively when `DataSourceStrategy` execution planning strategy is requested to [create a `RowDataSourceScanExec` physical operator \(possibly under `FilterExec` and `ProjectExec` operators\)](#) (which is when `DataSourceStrategy` is [executed](#) and `pruneFilterProject`).

Translating Catalyst Expression Into Data Source Filter Predicate — `translateFilter` Method

```
translateFilter(predicate: Expression): Option[Filter]
```

`translateFilter` translates a [Catalyst expression](#) into a corresponding [Filter predicate](#) if possible. If not, `translateFilter` returns `None`.

Table 2. `translateFilter`'s Conversions

Catalyst Expression	Filter Predicate
<code>EqualTo</code>	<code>EqualTo</code>
<code>EqualNullSafe</code>	<code>EqualNullSafe</code>
<code>GreaterThan</code>	<code>GreaterThan</code>
<code>LessThan</code>	<code>LessThan</code>
<code>GreaterThanOrEqual</code>	<code>GreaterThanOrEqual</code>
<code>LessThanOrEqual</code>	<code>LessThanOrEqual</code>
<code>InSet</code>	<code>In</code>
<code>In</code>	<code>In</code>
<code>IsNull</code>	<code>IsNull</code>
<code>IsNotNull</code>	<code>IsNotNull</code>
<code>And</code>	<code>And</code>
<code>Or</code>	<code>Or</code>
<code>Not</code>	<code>Not</code>
<code>StartsWith</code>	<code>StringStartsWith</code>
<code>EndsWith</code>	<code>StringEndsWith</code>
<code>Contains</code>	<code>StringContains</code>

Note

The Catalyst expressions and their corresponding data source filter predicates have the same names *in most cases* but belong to different Scala packages, i.e. `org.apache.spark.sql.catalyst.expressions` and `org.apache.spark.sql.sources`, respectively.

Note

- `translateFilter` is used when:
- `FileSourceScanExec` is created (and initializes `pushedDownFilters`)
 - `DataSourceStrategy` is requested to `selectFilters`
 - `PushDownOperatorsToDataSource` logical optimization is executed (for `DataSourceV2Relation` leaf operators with a `SupportsPushDownFilters` data source reader)

RDD Conversion (Converting RDD of Rows to Catalyst RDD of InternalRows) — `toCatalystRDD` Internal Method

```
toCatalystRDD(
    relation: LogicalRelation,
    output: Seq[Attribute],
    rdd: RDD[Row]): RDD[InternalRow]
toCatalystRDD(relation: LogicalRelation, rdd: RDD[Row]) (1)
```

1. Calls the former `toCatalystRDD` with the `output` of the `LogicalRelation`

`toCatalystRDD` branches off per the `needConversion` flag of the `BaseRelation` of the input `LogicalRelation`.

When enabled (`true`), `toCatalystRDD` converts the objects inside Rows to Catalyst types.

Note

`needConversion` flag is enabled (`true`) by default.

Otherwise, `toCatalystRDD` simply casts the input `RDD[Row]` to a `RDD[InternalRow]` (as a simple untyped Scala type conversion using Java's `asInstanceOf` operator).

Note

`toCatalystRDD` is used when `DataSourceStrategy` execution planning strategy is executed (for all kinds of `BaseRelations`).

Creating RowDataSourceScanExec Physical Operator for LogicalRelation (Possibly Under FilterExec and ProjectExec Operators) — `pruneFilterProjectRaw` Internal Method

```
pruneFilterProjectRaw(
    relation: LogicalRelation,
    projects: Seq[NamedExpression],
    filterPredicates: Seq[Expression],
    scanBuilder: (Seq[Attribute], Seq[Expression], Seq[Filter]) => RDD[InternalRow]): SparkPlan
```

`pruneFilterProjectRaw` creates a [RowDataSourceScanExec](#) leaf physical operator given a [LogicalRelation](#) leaf logical operator (possibly as a child of a [FilterExec](#) and a [ProjectExec](#) unary physical operators).

In other words, `pruneFilterProjectRaw` simply converts a [LogicalRelation](#) leaf logical operator into a [RowDataSourceScanExec](#) leaf physical operator (possibly under a [FilterExec](#) and a [ProjectExec](#) unary physical operators).

Note	<code>pruneFilterProjectRaw</code> is almost like SparkPlanner.pruneFilterProject .
------	---

Internally, `pruneFilterProjectRaw` splits the input `filterPredicates` expressions to [select the Catalyst expressions that can be converted to data source filter predicates](#) (and handled by the [BaseRelation](#) of the `LogicalRelation`).

`pruneFilterProjectRaw` combines all expressions that are neither convertible to data source filters nor can be handled by the relation using `And` binary expression (that creates a so-called `filterCondition` that will eventually be used to create a [FilterExec](#) physical operator if non-empty).

`pruneFilterProjectRaw` creates a [RowDataSourceScanExec](#) leaf physical operator.

If it is possible to use a column pruning only to get the right projection and if the columns of this projection are enough to evaluate all filter conditions, `pruneFilterProjectRaw` creates a [FilterExec](#) unary physical operator (with the unhandled predicate expressions and the [RowDataSourceScanExec](#) leaf physical operator as the child).

Note	In this case no extra ProjectExec unary physical operator is created.
------	---

Otherwise, `pruneFilterProjectRaw` creates a [FilterExec](#) unary physical operator (with the unhandled predicate expressions and the [RowDataSourceScanExec](#) leaf physical operator as the child) that in turn becomes the [child](#) of a new [ProjectExec](#) unary physical operator.

Note	<code>pruneFilterProjectRaw</code> is used exclusively when <code>DataSourceStrategy</code> execution planning strategy is executed (for a <code>LogicalRelation</code> with a <code>CatalystScan</code> relation) and pruneFilterProject (when executed for a <code>LogicalRelation</code> with a <code>PrunedFilteredScan</code> or a <code>PrunedScan</code> relation).
------	--

DataSourceV2Strategy Execution Planning Strategy

`DataSourceV2Strategy` is an [execution planning strategy](#) that [Spark Planner](#) uses to [plan](#) logical operators (from the [Data Source API V2](#)).

Table 1. `DataSourceV2Strategy`'s Execution Planning

Logical Operator	Physical Operator
<code>DataSourceV2Relation</code>	<code>DataSourceV2ScanExec</code>
<code>StreamingDataSourceV2Relation</code>	<code>DataSourceV2ScanExec</code>
<code>WriteToDataSourceV2</code>	<code>WriteToDataSourceV2Exec</code>
<code>AppendData</code> with <code>DataSourceV2Relation</code>	<code>WriteToDataSourceV2Exec</code>
<code>WriteToContinuousDataSource</code>	<code>WriteToContinuousDataSourceExec</code>
<code>Repartition</code> with a <code>StreamingDataSourceV2Relation</code> and a <code>ContinuousReader</code>	<code>ContinuousCoalesceExec</code>

Tip	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.sql.execution.datasources.v2.DataSourceV2Strategy</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.datasources.v2.DataSourceV2Strategy=</pre> <p>Refer to Logging.</p>
------------	---

Applying `DataSourceV2Strategy` Strategy to Logical Plan (Executing `DataSourceV2Strategy`) — `apply` Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

Note	<p><code>apply</code> is part of GenericStrategy Contract to generate a collection of <code>SparkPlans</code> for a given <code>logical plan</code>.</p>
-------------	--

`apply` branches off per the given [logical operator](#).

DataSourceV2Relation Logical Operator

For a [DataSourceV2Relation](#) logical operator, `apply` requests the [DataSourceV2Relation](#) for the [DataSourceReader](#).

`apply` then [pushFilters](#) followed by [pruneColumns](#).

`apply` prints out the following INFO message to the logs:

```
Pushing operators to [ClassName of DataSourceV2]
Pushed Filters: [pushedFilters]
Post-Scan Filters: [postScanFilters]
Output: [output]
```

`apply` uses the [DataSourceV2Relation](#) to create a [DataSourceV2ScanExec](#) physical operator.

If there are any [postScanFilters](#), `apply` creates a [FilterExec](#) physical operator with the [DataSourceV2ScanExec](#) physical operator as the child.

In the end, `apply` creates a [ProjectExec](#) physical operator with the [FilterExec](#) with the [DataSourceV2ScanExec](#) or directly with the [DataSourceV2ScanExec](#) physical operator.

StreamingDataSourceV2Relation Logical Operator

For a [StreamingDataSourceV2Relation](#) logical operator, `apply` ...FIXME

WriteToDataSourceV2 Logical Operator

For a [WriteToDataSourceV2](#) logical operator, `apply` simply creates a [WriteToDataSourceV2Exec](#) physical operator.

AppendData Logical Operator

For a [AppendData](#) logical operator with a [DataSourceV2Relation](#), `apply` requests the [DataSourceV2Relation](#) to create a [DataSourceWriter](#) that is used to create a [WriteToDataSourceV2Exec](#) physical operator.

WriteToContinuousDataSource Logical Operator

For a [writeToContinuousDataSource](#) logical operator, `apply` ...FIXME

Repartition Logical Operator

For a Repartition logical operator, apply ...FIXME

pushFilters Internal Method

```
pushFilters(  
    reader: DataSourceReader,  
    filters: Seq[Expression]): (Seq[Expression], Seq[Expression])
```

Note	pushFilters handles DataSourceReaders with SupportsPushDownFilters support only.
------	--

For the given `DataSourceReaders` with `SupportsPushDownFilters` support, `pushFilters` uses the `DataSourceStrategy` object to [translate every filter](#) in the given `filters`.

`pushFilters` requests the `SupportsPushDownFilters` reader to [pushFilters](#) first and then for the [pushedFilters](#).

In the end, `pushFilters` returns a pair of filters pushed and not.

Note	pushFilters is used exclusively when <code>DataSourceV2Strategy</code> execution planning strategy is executed (applied to a <code>DataSourceV2Relation</code> logical operator).
------	---

Column Pruning — pruneColumns Internal Method

```
pruneColumns(  
    reader: DataSourceReader,  
    relation: DataSourceV2Relation,  
    exprs: Seq[Expression]): Seq[AttributeReference]
```

`pruneColumns` ...FIXME

Note	<code>pruneColumns</code> is used when...FIXME
------	--

FileSourceStrategy Execution Planning Strategy for LogicalRelations with HadoopFsRelation

`FileSourceStrategy` is an [execution planning strategy](#) that [plans scans over collections of files](#) (possibly partitioned or bucketed).

`FileSourceStrategy` is part of [predefined strategies](#) of the [Spark Planner](#).

```
import org.apache.spark.sql.execution.datasources.FileSourceStrategy

// Enable INFO logging level to see the details of the strategy
val logger = FileSourceStrategy.getClass.getName.replace("$", "")
import org.apache.log4j.{Level, Logger}
Logger.getLogger(logger).setLevel(Level.INFO)

// Create a bucketed data source table
val tableName = "bucketed_4_id"
spark
  .range(100)
  .write
  .bucketBy(4, "id")
  .sortBy("id")
  .mode("overwrite")
  .saveAsTable(tableName)
val q = spark.table(tableName)
val plan = q.queryExecution.optimizedPlan

val executionPlan = FileSourceStrategy(plan).head

scala> println(executionPlan.numberedTreeString)
00 FileScan parquet default.bucketed_4_id[id#140L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/Users/jacek/dev/apps/spark-2.3.0-bin-hadoop2.7/spark-workhouse/bucketed_4..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id: bigint>

import org.apache.spark.sql.execution.FileSourceScanExec
val scan = executionPlan.collectFirst { case fsse: FileSourceScanExec => fsse }.get

scala> :type scan
org.apache.spark.sql.execution.FileSourceScanExec
```

`FileSourceScanExec` supports [Bucket Pruning](#) for [LogicalRelations](#) over [HadoopFsRelation](#) with the [bucketing specification](#) with the following:

1. There is exactly one bucketing column

2. The number of buckets is greater than 1

```
// Using the table created above
// There is exactly one bucketing column, i.e. id
// The number of buckets is greater than 1, i.e. 4
val tableName = "bucketed_4_id"
val q = spark.table(tableName).where($"id" isin (50, 90))
val qe = q.queryExecution
val plan = qe.optimizedPlan
scala> println(optimizedPlan.numberedTreeString)
00 Filter id#7L IN (50,90)
01 +- Relation[id#7L] parquet

import org.apache.spark.sql.execution.datasources.FileSourceStrategy

// Enable INFO logging level to see the details of the strategy
val logger = FileSourceStrategy.getClass.getName.replace("$", "")
import org.apache.log4j.{Level, Logger}
Logger.getLogger(logger).setLevel(Level.INFO)

scala> val executionPlan = FileSourceStrategy(plan).head
18/11/18 17:56:53 INFO FileSourceStrategy: Pruning directories with:
18/11/18 17:56:53 INFO FileSourceStrategy: Pruned 2 out of 4 buckets.
18/11/18 17:56:53 INFO FileSourceStrategy: Post-Scan Filters: id#7L IN (50,90)
18/11/18 17:56:53 INFO FileSourceStrategy: Output Data Schema: struct<id: bigint>
18/11/18 17:56:53 INFO FileSourceScanExec: Pushed Filters: In(id, [50,90])
executionPlan: org.apache.spark.sql.execution.SparkPlan = ...

scala> println(executionPlan.numberedTreeString)
00 Filter id#7L IN (50,90)
01 +- FileScan parquet default.bucketed_4_id[id#7L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/Users/jacek/dev/oss/spark/spark-warehouse/bucketed_4_id], PartitionFilters: [], PushedFilters: [In(id, [50,90])], ReadSchema: struct<id:bigint>, SelectedBucketsCount: 2 out of 4
```

Tip	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.sql.execution.datasources.FileSourceStrategy</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.datasources.FileSourceStrategy=INFO</pre> <p>Refer to Logging.</p>
------------	---

collectProjectsAndFilters Method

```
collectProjectsAndFilters(plan: LogicalPlan):
  (Option[Seq[NamedExpression]], Seq[Expression], LogicalPlan, Map[Attribute, Expression])
```

`collectProjectsAndFilters` is a pattern used to destructure a [LogicalPlan](#) that can be `Project` or `Filter`. Any other `LogicalPlan` give an *all-empty* response.

Applying FileSourceStrategy Strategy to Logical Plan (Executing FileSourceStrategy) — apply Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

Note

`apply` is part of [GenericStrategy Contract](#) to generate a collection of [SparkPlans](#) for a given [logical plan](#).

`apply` uses [PhysicalOperation](#) Scala extractor object to destructure a logical query plan into a tuple of projection and filter expressions together with a leaf logical operator.

`apply` only works with [logical plans](#) that are actually a [LogicalRelation](#) with a [HadoopFsRelation](#) (possibly as a child of [Project](#) and [Filter](#) logical operators).

`apply` computes `partitionKeyFilters` expression set with the filter expressions that are a subset of the [partitionSchema](#) of the [HadoopFsRelation](#).

`apply` prints out the following INFO message to the logs:

```
Pruning directories with: [partitionKeyFilters]
```

`apply` computes `afterScanFilters` predicate [expressions](#) that should be evaluated after the scan.

`apply` prints out the following INFO message to the logs:

```
Post-Scan Filters: [afterScanFilters]
```

`apply` computes `readDataColumns` [attributes](#) that are the required attributes except the partition columns.

`apply` prints out the following INFO message to the logs:

```
Output Data Schema: [outputSchema]
```

`apply` creates a [FileSourceScanExec](#) physical operator.

If there are any `afterScanFilter` predicate expressions, `apply` creates a [FilterExec](#) physical operator with them and the `FileSourceScanExec` operator.

If the `output` of the `FilterExec` physical operator is different from the `projects` expressions, `apply` creates a [ProjectExec](#) physical operator with them and the `FilterExec` or the `FileSourceScanExec` operators.

HiveTableScans Execution Planning Strategy

`HiveTableScans` is an [execution planning strategy](#) (of [Hive-specific SparkPlanner](#)) that resolves [HiveTableRelation](#).

Applying HiveTableScans Strategy to Logical Plan (Executing HiveTableScans) — apply Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

Note

`apply` is part of [GenericStrategy Contract](#) to generate a collection of [SparkPlans](#) for a given [logical plan](#).

```
apply ...FIXME
```

InMemoryScans Execution Planning Strategy

`InMemoryScans` is an execution planning strategy that plans `InMemoryRelation` logical operators to `InMemoryTableScanExec` physical operators.

```
val spark: SparkSession = ...
// query uses InMemoryRelation logical operator
val q = spark.range(5).cache
val plan = q.queryExecution.optimizedPlan
scala> println(plan.numberedTreeString)
00 InMemoryRelation [id#208L], true, 10000, StorageLevel(disk, memory, deserialized, 1
replicas)
01   +- *Range (0, 5, step=1, splits=8)

// InMemoryScans is an internal class of SparkStrategies
import spark.sessionState.planner.InMemoryScans
val physicalPlan = InMemoryScans.apply(plan).head
scala> println(physicalPlan.numberedTreeString)
00 InMemoryTableScan [id#208L]
01   +- InMemoryRelation [id#208L], true, 10000, StorageLevel(disk, memory, deserialized,
1 replicas)
02         +- *Range (0, 5, step=1, splits=8)
```

`InMemoryScans` is part of the standard execution planning strategies of `SparkPlanner`.

Applying InMemoryScans Strategy to Logical Plan (Executing InMemoryScans) — `apply` Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

Note

`apply` is part of `GenericStrategy Contract` to generate a collection of `SparkPlans` for a given `logical plan`.

`apply` requests `PhysicalOperation` extractor to destructure the input logical plan to a `InMemoryRelation` logical operator.

In the end, `apply` `pruneFilterProject` with a new `InMemoryTableScanExec` physical operator.

JoinSelection Execution Planning Strategy

`JoinSelection` is an execution planning strategy that `SparkPlanner` uses to plan a `Join` logical operator to one of the supported join physical operators (as described by [join physical operator selection requirements](#)).

`JoinSelection` firstly [considers](#) join physical operators per whether join keys are used or not. When join keys are used, `JoinSelection` considers `BroadcastHashJoinExec`, `ShuffledHashJoinExec` or `SortMergeJoinExec` operators. Without join keys, `JoinSelection` considers `BroadcastNestedLoopJoinExec` or `CartesianProductExec`.

Table 1. Join Physical Operator Selection Requirements (in the order of preference)

Physical Join Operator	Selection Requirements
BroadcastHashJoinExec	<p>There are join keys and one of the following holds:</p> <ul style="list-style-type: none"> Join type is CROSS, INNER, LEFT ANTI, LEFT OUTER, LEFT SEMI or ExistenceJoin (i.e. canBuildRight for the input joinType is positive) and right join side can be broadcast Join type is CROSS, INNER or RIGHT OUTER (i.e. canBuildLeft for the input joinType is positive) and left join side can be broadcast
ShuffledHashJoinExec	<p>There are join keys and one of the following holds:</p> <ul style="list-style-type: none"> spark.sql.join.preferSortMergeJoin is disabled, the join type is CROSS, INNER, LEFT ANTI, LEFT OUTER, LEFT SEMI or ExistenceJoin (i.e. canBuildRight for the input joinType is positive), canBuildLocalHashMap for right join side and finally right join side is much smaller than left side spark.sql.join.preferSortMergeJoin is disabled, the join type is CROSS, INNER or RIGHT OUTER (i.e. canBuildLeft for the input joinType is positive), canBuildLocalHashMap for left join side and finally left join side is much smaller than right Left join keys are not orderable
SortMergeJoinExec	Left join keys are orderable
BroadcastNestedLoopJoinExec	<p>There are no join keys and one of the following holds:</p> <ul style="list-style-type: none"> Join type is CROSS, INNER, LEFT ANTI, LEFT OUTER, LEFT SEMI or ExistenceJoin (i.e. canBuildRight for the input joinType is positive) and right join side can be broadcast Join type is CROSS, INNER or RIGHT OUTER (i.e. canBuildLeft for the input joinType is positive) and left join side can be broadcast
CartesianProductExec	There are no join keys and join type is CROSS or INNER
BroadcastNestedLoopJoinExec	No other join operator has matched already

Note

`JoinSelection` uses `ExtractEquiJoinKeys` Scala extractor to destructure a `join` logical operator.

Is Left-Side Plan At Least 3 Times Smaller Than Right-Side Plan? — `muchSmaller` Internal Condition

```
muchSmaller(a: LogicalPlan, b: LogicalPlan): Boolean
```

`muchSmaller` condition holds when plan `a` is at least 3 times smaller than plan `b`.

Internally, `muchSmaller` calculates the estimated statistics for the input logical plans and compares their physical size in bytes (`sizeInBytes`).

Note

`muchSmaller` is used when `JoinSelection` checks join selection requirements for `ShuffledHashJoinExec` physical operator.

`canBuildLocalHashMap` Internal Condition

```
canBuildLocalHashMap(plan: LogicalPlan): Boolean
```

`canBuildLocalHashMap` condition holds for the logical `plan` whose single partition is small enough to build a hash table (i.e. `spark.sql.autoBroadcastJoinThreshold` multiplied by `spark.sql.shuffle.partitions`).

Internally, `canBuildLocalHashMap` calculates the estimated statistics for the input logical plans and takes the size in bytes (`sizeInBytes`).

Note

`canBuildLocalHashMap` is used when `JoinSelection` checks join selection requirements for `ShuffledHashJoinExec` physical operator.

Can Logical Plan Be Broadcast? — `canBroadcast` Internal Condition

```
canBroadcast(plan: LogicalPlan): Boolean
```

`canBroadcast` is enabled, i.e. `true`, when the size of the output of the input logical plan (aka `sizeInBytes`) is less than `spark.sql.autoBroadcastJoinThreshold` configuration property.

Note

`spark.sql.autoBroadcastJoinThreshold` is 10M by default.

Note	<code>canBroadcast</code> uses the total size statistic from Statistics of a logical operator.
Note	<code>canBroadcast</code> is used when <code>JoinSelection</code> is requested to canBroadcastBySizes and selects the build side per join type and total size statistic of join sides.

canBroadcastByHints Internal Method

```
canBroadcastByHints(joinType: JoinType, left: LogicalPlan, right: LogicalPlan): Boolean
```

`canBroadcastByHints` is positive (i.e. `true`) when either condition holds:

1. Join type is [CROSS](#), [INNER](#) or [RIGHT OUTER](#) (i.e. [canBuildLeft](#) for the input `joinType` is positive) and `left` operator's [broadcast](#) hint flag is on
2. Join type is [CROSS](#), [INNER](#), [LEFT ANTI](#), [LEFT OUTER](#), [LEFT SEMI](#) or [ExistenceJoin](#) (i.e. [canBuildRight](#) for the input `joinType` is positive) and `right` operator's [broadcast](#) hint flag is on

Otherwise, `canBroadcastByHints` is negative (i.e. `false`).

Note `canBroadcastByHints` is used when `JoinSelection` is requested to [plan a Join logical operator](#) (and considers a [BroadcastHashJoinExec](#) or a [BroadcastNestedLoopJoinExec](#) physical operator).

Selecting Build Side Per Join Type and Broadcast Hints — broadcastSideByHints Internal Method

```
broadcastSideByHints(joinType: JoinType, left: LogicalPlan, right: LogicalPlan): Build Side
```

`broadcastSideByHints` computes `buildLeft` and `buildRight` flags:

- `buildLeft` flag is positive (i.e. `true`) when the join type is [CROSS](#), [INNER](#) or [RIGHT OUTER](#) (i.e. [canBuildLeft](#) for the input `joinType` is positive) and the `left` operator's [broadcast](#) hint flag is positive
- `buildRight` flag is positive (i.e. `true`) when the join type is [CROSS](#), [INNER](#), [LEFT ANTI](#), [LEFT OUTER](#), [LEFT SEMI](#) or [ExistenceJoin](#) (i.e. [canBuildRight](#) for the input `joinType` is positive) and the `right` operator's [broadcast](#) hint flag is positive

In the end, `broadcastSideByHints` gives the join side to broadcast.

Note

`broadcastSideByHints` is used when `JoinSelection` is requested to plan a Join logical operator (and considers a `BroadcastHashJoinExec` or a `BroadcastNestedLoopJoinExec` physical operator).

Choosing Join Side to Broadcast— `broadcastSide` Internal Method

```
broadcastSide(  
    canBuildLeft: Boolean,  
    canBuildRight: Boolean,  
    left: LogicalPlan,  
    right: LogicalPlan): BuildSide
```

`broadcastSide` gives the smaller side (`BuildRight` or `BuildLeft`) per total size when `canBuildLeft` and `canBuildRight` are both positive (i.e. `true`).

`broadcastSide` gives `BuildRight` when `canBuildRight` is positive.

`broadcastSide` gives `BuildLeft` when `canBuildLeft` is positive.

When all the above conditions are not met, `broadcastSide` gives the smaller side (`BuildRight` or `BuildLeft`) per total size (similarly to the first case when `canBuildLeft` and `canBuildRight` are both positive).

Note

`broadcastSide` is used when `JoinSelection` is requested to `broadcastSideByHints`, select the build side per join type and total size statistic of join sides, and `execute` (and considers a `BroadcastNestedLoopJoinExec` physical operator).

Checking If Join Type Allows For Left Join Side As Build Side— `canBuildLeft` Internal Condition

```
canBuildLeft(joinType: JoinType): Boolean
```

`canBuildLeft` is positive (i.e. `true`) for `CROSS`, `INNER` and `RIGHT OUTER` join types. Otherwise, `canBuildLeft` is negative (i.e. `false`).

Note

`canBuildLeft` is used when `JoinSelection` is requested to `canBroadcastByHints`, `broadcastSideByHints`, `canBroadcastBySizes`, `broadcastSideBySizes` and `execute` (when selecting a `[ShuffledHashJoinExec]` physical operator).

Checking If Join Type Allows For Right Join Side As Build Side — `canBuildRight` Internal Condition

```
canBuildRight(joinType: JoinType): Boolean
```

`canBuildRight` is positive (i.e. `true`) if the input join type is one of the following:

- `CROSS`, `INNER`, `LEFT ANTI`, `LEFT OUTER`, `LEFT SEMI` or `ExistenceJoin`

Otherwise, `canBuildRight` is negative (i.e. `false`).

Note

`canBuildRight` is used when `JoinSelection` is requested to `canBroadcastByHints`, `broadcastSideByHints`, `canBroadcastBySizes`, `broadcastSideBySizes` and `execute` (when selecting a `[ShuffledHashJoinExec]` physical operator).

Checking If Join Type and Total Size Statistic of Join Sides Allow for Broadcast Join — `canBroadcastBySizes` Internal Method

```
canBroadcastBySizes(joinType: JoinType, left: LogicalPlan, right: LogicalPlan): Boolean
```

`canBroadcastBySizes` is positive (i.e. `true`) when either condition holds:

1. Join type is `CROSS`, `INNER` or `RIGHT OUTER` (i.e. `canBuildLeft` for the input `joinType` is positive) and `left` operator `can be broadcast per total size statistic`
2. Join type is `CROSS`, `INNER`, `LEFT ANTI`, `LEFT OUTER`, `LEFT SEMI` or `ExistenceJoin` (i.e. `canBuildRight` for the input `joinType` is positive) and `right` operator `can be broadcast per total size statistic`

Otherwise, `canBroadcastByHints` is negative (i.e. `false`).

Note

`canBroadcastByHints` is used when `JoinSelection` is requested to `plan a Join logical operator` (and considers a `BroadcastHashJoinExec` or a `BroadcastNestedLoopJoinExec` physical operator).

Selecting Build Side Per Join Type and Total Size Statistic of Join Sides — `broadcastSideBySizes` Internal Method

```
broadcastSideBySizes(joinType: JoinType, left: LogicalPlan, right: LogicalPlan): Build
Side
```

`broadcastSideBySizes` computes `buildLeft` and `buildRight` flags:

- `buildLeft` flag is positive (i.e. `true`) when the join type is **CROSS**, **INNER** or **RIGHT OUTER** (i.e. `canBuildLeft` for the input `joinType` is positive) and `left` operator can be broadcast per total size statistic
- `buildRight` flag is positive (i.e. `true`) when the join type is **CROSS**, **INNER**, **LEFT ANTI**, **LEFT OUTER**, **LEFT SEMI** or **ExistenceJoin** (i.e. `canBuildRight` for the input `joinType` is positive) and `right` operator can be broadcast per total size statistic

In the end, `broadcastSideByHints` gives the join side to broadcast.

Note

`broadcastSideByHints` is used when `JoinSelection` is requested to plan a Join logical operator (and considers a `BroadcastHashJoinExec` or a `BroadcastNestedLoopJoinExec` physical operator).

Applying JoinSelection Strategy to Logical Plan (Executing JoinSelection) — apply Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

Note

`apply` is part of **GenericStrategy Contract** to generate a collection of **SparkPlans** for a given logical plan.

`apply` uses `ExtractEquiJoinKeys` Scala extractor to destructure the input logical `plan`.

Considering BroadcastHashJoinExec Physical Operator

`apply` gives a `BroadcastHashJoinExec` physical operator if the plan **should be broadcast per join type and broadcast hints used** (for the join type and left or right side of the join). `apply` selects the build side per join type and broadcast hints.

`apply` gives a `BroadcastHashJoinExec` physical operator if the plan **should be broadcast per join type and size of join sides** (for the join type and left or right side of the join). `apply` selects the build side per join type and total size statistic of join sides.

Considering ShuffledHashJoinExec Physical Operator

`apply` gives...FIXME

Considering SortMergeJoinExec Physical Operator

apply gives...FIXME

Considering BroadcastNestedLoopJoinExec Physical Operator

apply gives...FIXME

Considering CartesianProductExec Physical Operator

apply gives...FIXME

SpecialLimits Execution Planning Strategy

`SpecialLimits` is an [execution planning strategy](#) that [Spark Planner](#) uses to [FIXME](#).

Applying SpecialLimits Strategy to Logical Plan (Executing SpecialLimits) — `apply` Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

Note

`apply` is part of [GenericStrategy Contract](#) to generate a collection of [SparkPlans](#) for a given [logical plan](#).

`apply ...FIXME`

CollapseCodegenStages Physical Query Optimization — Collapsing Physical Operators for Whole-Stage Java Code Generation (aka Whole-Stage CodeGen)

`CollapseCodegenStages` is a **physical query optimization** (aka *physical query preparation rule* or simply *preparation rule*) that collapses physical operators and generates a Java source code for their execution.

When `executed` (with whole-stage code generation enabled), `CollapseCodegenStages` inserts `WholeStageCodegenExec` or `InputAdapter` physical operators to a physical plan.

`CollapseCodegenStages` uses so-called **control gates** before deciding whether a **physical operator** supports the **whole-stage Java code generation** or not (and what physical operator to insert):

1. Factors in physical operators with `CodeGenSupport` only
2. Enforces the `supportCodegen` custom requirements on a physical operator, i.e.
 - i. `supportCodegen` flag turned on (`true`)
 - ii. No Catalyst expressions are `CodegenFallback`
 - iii. Output schema is **neither wide nor deep** and uses just enough fields (including nested fields)
 - iv. Children use output schema that is also **neither wide nor deep**

Note

`spark.sqlcodegen.maxFields` Spark internal property controls the total number of fields in a schema that is acceptable for whole-stage code generation.

The number is `100` by default.

Technically, `CollapseCodegenStages` is just a **Catalyst rule** for transforming **physical query plans**, i.e. `Rule[SparkPlan]`.

`CollapseCodegenStages` is part of **preparations** batch of physical query plan rules and is executed when `QueryExecution` is requested for the **optimized physical query plan** (i.e. in `executedPlan` phase of a query execution).

```
val q = spark.range(3).groupBy('id % 2 as "gid").count
// Let's see where and how many "stars" does this query get
```

```

scala> q.explain
== Physical Plan ==
*(2) HashAggregate(keys=[(id#0L % 2)#9L], functions=[count(1)])
+- Exchange hashpartitioning((id#0L % 2)#9L, 200)
  +- *(1) HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#9L], functions=[partial_count(1)])
    +- *(1) Range (0, 3, step=1, splits=8)

// There are two stage IDs: 1 and 2 (see the round brackets)
// Looks like Exchange physical operator does not support codegen
// Let's walk through the query execution phases and see it ourselves

// sparkPlan phase is just before CollapseCodegenStages physical optimization is applied
val sparkPlan = q.queryExecution.sparkPlan
scala> println(sparkPlan.numberedTreeString)
00 HashAggregate(keys=[(id#0L % 2)#12L], functions=[count(1)], output=[gid#2L, count#5L])
01 +- HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#12L], functions=[partial_count(1)], output=[(id#0L % 2)#12L, count#11L])
02   +- Range (0, 3, step=1, splits=8)

// Compare the above with the executedPlan phase
// which happens to be after CollapseCodegenStages physical optimization
scala> println(q.queryExecution.executedPlan.numberedTreeString)
00 *(2) HashAggregate(keys=[(id#0L % 2)#12L], functions=[count(1)], output=[gid#2L, count#5L])
01 +- Exchange hashpartitioning((id#0L % 2)#12L, 200)
02   +- *(1) HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#12L], functions=[partial_count(1)], output=[(id#0L % 2)#12L, count#11L])
03     +- *(1) Range (0, 3, step=1, splits=8)

// Let's apply the CollapseCodegenStages rule ourselves
import org.apache.spark.sql.execution.CollapseCodegenStages
val ccsRule = CollapseCodegenStages(spark.sessionState.conf)
scala> val planAfterCCS = ccsRule.apply(sparkPlan)
planAfterCCS: org.apache.spark.sql.execution.SparkPlan =
*(1) HashAggregate(keys=[(id#0L % 2)#12L], functions=[count(1)], output=[gid#2L, count#5L])
+- *(1) HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#12L], functions=[partial_count(1)], output=[(id#0L % 2)#12L, count#11L])
  +- *(1) Range (0, 3, step=1, splits=8)

// The number of stage IDs do not match
// Looks like the above misses one or more rules
// EnsureRequirements optimization rule?
// It is indeed executed before CollapseCodegenStages
import org.apache.spark.sql.execution.exchange.EnsureRequirements
val erRule = EnsureRequirements(spark.sessionState.conf)
val planAfterER = erRule.apply(sparkPlan)
scala> println(planAfterER.numberedTreeString)
00 HashAggregate(keys=[(id#0L % 2)#12L], functions=[count(1)], output=[gid#2L, count#5L])

```

```

01 +- Exchange hashpartitioning((id#0L % 2)#12L, 200)
02   +- HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#12L], functions=[partial_count(
03     1)], output=[(id#0L % 2)#12L, count#11L])
04     +- Range (0, 3, step=1, splits=8)

// Time for CollapseCodegenStages
val planAfterCCS = ccsRule.apply(planAfterER)
scala> println(planAfterCCS.numberedTreeString)
00 *(2) HashAggregate(keys=[(id#0L % 2)#12L], functions=[count(1)], output=[gid#2L, co
unt#5L])
01 +- Exchange hashpartitioning((id#0L % 2)#12L, 200)
02   +- *(1) HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#12L], functions=[partial_
count(1)], output=[(id#0L % 2)#12L, count#11L])
03     +- *(1) Range (0, 3, step=1, splits=8)

assert(planAfterCCS == q.queryExecution.executedPlan, "Plan after ER and CCS rules sho
uld match the executedPlan plan")

// Bingo!
// The result plan matches the executedPlan plan

// HashAggregateExec and Range physical operators support codegen (is a CodegenSupport)

// - HashAggregateExec disables codegen for ImperativeAggregate aggregate functions
// ShuffleExchangeExec does not support codegen (is not a CodegenSupport)

// The top-level physical operator should be WholeStageCodegenExec
import org.apache.spark.sql.execution.WholeStageCodegenExec
val wsce = planAfterCCS.asInstanceOf[WholeStageCodegenExec]

// The single child operator should be HashAggregateExec
import org.apache.spark.sql.execution.aggregate.HashAggregateExec
val hae = wsce.child.asInstanceOf[HashAggregateExec]

// Since ShuffleExchangeExec does not support codegen, the child of HashAggregateExec
// is InputAdapter
import org.apache.spark.sql.execution.InputAdapter
val ia = hae.child.asInstanceOf[InputAdapter]

// And it's only now when we can get at ShuffleExchangeExec
import org.apache.spark.sql.execution.exchange.ShuffleExchangeExec
val se = ia.child.asInstanceOf[ShuffleExchangeExec]

```

With `spark.sql.codegen.wholeStage` Spark internal property enabled (which is on by default), `CollapseCodegenStages` finds physical operators with `CodegenSupport` for which `whole-stage codegen requirements hold` and collapses them together as `WholeStageCodegenExec` physical operator (possibly with `InputAdapter` in-between for physical operators with no support for Java code generation).

Note

`InputAdapter` shows itself with no star in the output of `explain` (or `TreeNode.numberedTreeString`).

```
val q = spark.range(1).groupBy("id").count
scala> q.explain
== Physical Plan ==
*HashAggregate(keys=[id#16L], functions=[count(1)])
+- Exchange hashpartitioning(id#16L, 200)
  +- *HashAggregate(keys=[id#16L], functions=[partial_count(1)])
    +- *Range (0, 1, step=1, splits=8)
```

`CollapseCodegenStages` takes a [SQLConf](#) when created.

Note

You can disable `collapseCodegenStages` (and so whole-stage Java code generation) by turning `spark.sqlcodegen.wholeStage` Spark internal property off.

`spark.sqlcodegen.wholeStage` property is enabled by default.

```
import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_CODEGEN_ENABLED
scala> spark.conf.get(WHOLESTAGE_CODEGEN_ENABLED)
res0: String = true
```

Use `SQLConf.wholeStageEnabled` method to access the current value.

```
scala> spark.sessionState.conf.wholeStageEnabled
res1: Boolean = true
```

Tip

Import `CollapseCodegenStages` and apply the rule directly to a physical plan to learn how the rule works.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
// Just a structured query with explode Generator expression that supports codegen "partially"
// i.e. explode extends CodegenSupport but codegenSupport flag is off
val q = spark.range(2)
  .filter($"id" === 0)
  .select(explode(lit(Array(0,1,2))) as "exploded")
  .join(spark.range(2))
  .where($"exploded" === $"id")
scala> q.show
+-----+---+
|exploded| id|
+-----+---+
|        0|  0|
|        1|  1|
+-----+---+
```

```

// the final physical plan (after CollapseCodegenStages applied and the other optimization rules)
scala> q.explain
== Physical Plan ==
*BroadcastHashJoin [cast(exploded#34 as bigint)], [id#37L], Inner, BuildRight
:- *Filter isnotnull(exploded#34)
: +- Generate explode([0,1,2]), false, false, [exploded#34]
:   +- *Project
:     +- *Filter (id#29L = 0)
:       +- *Range (0, 2, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 2, step=1, splits=8)

// Control when CollapseCodegenStages is applied to a query plan
// Take sparkPlan that is a physical plan before optimizations, incl. CollapseCodegenStages
val plan = q.queryExecution.sparkPlan

// Is wholeStageEnabled enabled?
// It is by default
scala> println(spark.sessionState.conf.wholeStageEnabled)
true

import org.apache.spark.sql.execution.CollapseCodegenStages
val ccs = CollapseCodegenStages(conf = spark.sessionState.conf)

scala> ccs.ruleName
res0: String = org.apache.spark.sql.execution.CollapseCodegenStages

// Before CollapseCodegenStages
scala> println(plan.numberedTreeString)
00 BroadcastHashJoin [cast(exploded#34 as bigint)], [id#37L], Inner, BuildRight
01 :- Filter isnotnull(exploded#34)
02 : +- Generate explode([0,1,2]), false, false, [exploded#34]
03 :   +- Project
04 :     +- Filter (id#29L = 0)
05 :       +- Range (0, 2, step=1, splits=8)
06 +- Range (0, 2, step=1, splits=8)

// After CollapseCodegenStages
// Note the stars (that WholeStageCodegenExec.generateTreeString gives)
val execPlan = ccs.apply(plan)
scala> println(execPlan.numberedTreeString)
00 *BroadcastHashJoin [cast(exploded#34 as bigint)], [id#37L], Inner, BuildRight
01 :- *Filter isnotnull(exploded#34)
02 : +- Generate explode([0,1,2]), false, false, [exploded#34]
03 :   +- *Project
04 :     +- *Filter (id#29L = 0)
05 :       +- *Range (0, 2, step=1, splits=8)
06 +- *Range (0, 2, step=1, splits=8)

// The first star is from WholeStageCodegenExec physical operator

```

```

import org.apache.spark.sql.execution.WholeStageCodegenExec
val wsc = execPlan(0).asInstanceOf[WholeStageCodegenExec]
scala> println(wsc.numberedTreeString)
00 *BroadcastHashJoin [cast(exploded#34 as bigint)], [id#37L], Inner, BuildRight
01 :- *Filter isnotnull(exploded#34)
02 : +- Generate explode([0,1,2]), false, false, [exploded#34]
03 :     +- *Project
04 :         +- *Filter (id#29L = 0)
05 :             +- *Range (0, 2, step=1, splits=8)
06 +- *Range (0, 2, step=1, splits=8)

// Let's disable wholeStage codegen
// CollapseCodegenStages becomes a noop
// It is as if we were not applied Spark optimizations to a physical plan
// We're selective as we only disable whole-stage codegen
val newSpark = spark.newSession()
import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_CODEGEN_ENABLED
newSpark.sessionState.conf.setConf(WHOLESTAGE_CODEGEN_ENABLED, false)
scala> println(newSpark.sessionState.conf.wholeStageEnabled)
false

// Whole-stage codegen is disabled
// So regardless whether you do apply Spark optimizations or not
// Java code generation won't take place
val ccsWholeStageDisabled = CollapseCodegenStages(conf = newSpark.sessionState.conf)
val execPlan = ccsWholeStageDisabled.apply(plan)
// Note no stars in the output
scala> println(execPlan.numberedTreeString)
00 BroadcastHashJoin [cast(exploded#34 as bigint)], [id#37L], Inner, BuildRight
01 :- Filter isnotnull(exploded#34)
02 : +- Generate explode([0,1,2]), false, false, [exploded#34]
03 :     +- Project
04 :         +- Filter (id#29L = 0)
05 :             +- Range (0, 2, step=1, splits=8)
06 +- Range (0, 2, step=1, splits=8)

```

Executing Rule (Inserting WholeStageCodegenExec or InputAdapter into Physical Query Plan for Whole-Stage Java Code Generation) — apply Method

```
apply(plan: SparkPlan): SparkPlan
```

Note

`apply` is part of the [Rule Contract](#) to apply a rule to a [TreeNode](#) (e.g. [physical plan](#)).

`apply` starts [inserting WholeStageCodegenExec \(with InputAdapter\)](#) in the input `plan` physical plan only when [spark.sql.codegen.wholeStage](#) Spark internal property is turned on.

Otherwise, `apply` does nothing at all (i.e. passes the input physical plan through unchanged).

Inserting WholeStageCodegenExec Physical Operator For Codegen Stages — `insertWholeStageCodegen` Internal Method

```
insertWholeStageCodegen(plan: SparkPlan): SparkPlan
```

Internally, `insertWholeStageCodegen` branches off per [physical operator](#):

1. For physical operators with a single [output schema attribute](#) of type `ObjectType`, `insertWholeStageCodegen` requests the operator for the `child` physical operators and tries to [insertWholeStageCodegen](#) on them only.
1. For physical operators that support [Java code generation](#) and meets the [additional requirements for codegen](#), `insertWholeStageCodegen` [insertInputAdapter](#) (with the operator), requests `WholeStageCodegenId` for the `getNextStageId` and then uses both to return a new [WholeStageCodegenExec](#) physical operator.
2. For any other physical operators, `insertWholeStageCodegen` requests the operator for the `child` physical operators and tries to [insertWholeStageCodegen](#) on them only.

```
// FIXME: DEMO
// Step 1. The top-level physical operator is CodegenSupport with supportCodegen enabled
// Step 2. The top-level operator is CodegenSupport with supportCodegen disabled
// Step 3. The top-level operator is not CodegenSupport
// Step 4. "plan.output.length == 1 && plan.output.head.dataType.asInstanceOf[ObjectType]"
```

Note

`insertWholeStageCodegen` explicitly skips physical operators with a single-attribute [output schema](#) with the type of the attribute being `ObjectType` type.

Note

`insertWholeStageCodegen` is used recursively when `CollapseCodegenStages` is requested for the following:

- [Executes](#) (and walks down a physical plan)
- [Inserts InputAdapter physical operator](#)

Inserting InputAdapter Unary Physical Operator — `insertInputAdapter` Internal Method

```
insertInputAdapter(plan: SparkPlan): SparkPlan
```

`insertInputAdapter` inserts an [InputAdapter](#) physical operator in a physical plan.

- For [SortMergeJoinExec](#) (with inner and outer joins) [inserts an InputAdapter operator](#) for both children physical operators individually
- For [codegen-unsupported](#) operators [inserts an InputAdapter operator](#)
- For other operators (except [SortMergeJoinExec](#) operator above or for which [Java code cannot be generated](#)) [inserts a WholeStageCodegenExec operator](#) for every child operator

Caution	FIXME Examples for every case + screenshots from web UI
---------	---

Note	<code>insertInputAdapter</code> is used exclusively when <code>CollapseCodegenStages</code> inserts WholeStageCodegenExec physical operator and recursively down the physical plan.
------	---

Enforcing Whole-Stage CodeGen Requirements For Physical Operators — `supportCodegen` Internal Predicate

```
supportCodegen(plan: SparkPlan): Boolean
```

`supportCodegen` is positive (`true`) when the input [physical operator](#) is as follows:

1. [CodegenSupport](#) and the [supportCodegen](#) flag is turned on

Note	<code>supportCodegen</code> flag is turned on by default.
------	---

2. No [Catalyst](#) expressions are [CodegenFallback](#) (except [LeafExpressions](#))
3. Output schema is **neither wide nor deep**, i.e. [uses just enough fields](#) (including nested fields)

Note	<code>spark.sql.codegen.maxFields</code> Spark internal property defaults to <code>100</code> .
------	---

4. Children also have the output schema that is [neither wide nor deep](#)

Otherwise, `supportCodegen` is negative (`false`).

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
// both where and select operators support codegen
// the plan tree (with the operators and expressions) meets the requirements
// That's why the plan has WholeStageCodegenExec inserted
// That you can see as stars (*) in the output of explain
val q = Seq((1,2,3)).toDF("id", "c0", "c1").where('id === 0).select('c0)
scala> q.explain
== Physical Plan ==
*Project [_2#89 AS c0#93]
+- *Filter (_1#88 = 0)
   +- LocalTableScan [_1#88, _2#89, _3#90]

// CollapseCodegenStages is only used in QueryExecution.executedPlan
// Use sparkPlan then so we avoid CollapseCodegenStages
val plan = q.queryExecution.sparkPlan
import org.apache.spark.sql.execution.ProjectExec
val pe = plan.asInstanceOf[ProjectExec]

scala> pe.supportCodegen
res1: Boolean = true

scala> pe.schema.fields.size
res2: Int = 1

scala> pe.children.map(_.schema).map(_.size).sum
res3: Int = 3
```

```

import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
// both where and select support codegen
// let's break the requirement of spark.sqlcodegen.maxFields
val newSpark = spark.newSession()
import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_MAX_NUM_FIELDS
newSpark.sessionState.conf.setConf(WHOLESTAGE_MAX_NUM_FIELDS, 2)

scala> println(newSpark.sessionState.conf.wholeStageMaxNumFields)
2

import newSpark.implicits._
// the same query as above but created in SparkSession with WHOLESTAGE_MAX_NUM_FIELDS
as 2
val q = Seq((1,2,3)).toDF("id", "c0", "c1").where('id === 0).select('c0)

// Note that there are no stars in the output of explain
// No WholeStageCodegenExec operator in the plan => whole-stage codegen disabled
scala> q.explain
== Physical Plan ==
Project [_2#122 AS c0#126]
+- Filter (_1#121 = 0)
   +- LocalTableScan [_1#121, _2#122, _3#123]

```

Note

`supportCodegen` is used when `CollapseCodegenStages` does the following:

- Inserts [InputAdapter physical operator](#) for physical plans that do not support whole-stage Java code generation (i.e. `supportCodegen` is turned off).
- Inserts [WholeStageCodegenExec physical operator](#) for physical operators that do support whole-stage Java code generation (i.e. `supportCodegen` is turned on).

Enforcing Whole-Stage CodeGen Requirements For Catalyst Expressions — `supportCodegen` Internal Predicate

```
supportCodegen(e: Expression): Boolean
```

`supportCodegen` is positive (`true`) when the input [Catalyst expression](#) is the following (in the order of verification):

1. [LeafExpression](#)
2. [non-CodegenFallback](#)

Otherwise, `supportCodegen` is negative (`false`).

Note

`supportCodegen` (for [Catalyst expressions](#)) is used exclusively when `collapseCodegenStages` physical optimization is requested to [enforce whole-stage codegen requirements for a physical operator](#).

EnsureRequirements Physical Query Optimization

`EnsureRequirements` is a **physical query optimization** (aka *physical query preparation rule* or simply *preparation rule*) that `QueryExecution` uses to optimize the physical plan of a structured query by transforming the following physical operators (up the plan tree):

1. Removes two adjacent `ShuffleExchangeExec` physical operators if the child partitioning scheme guarantees the parent's partitioning
2. For other non- `shuffleExchangeExec` physical operators, ensures partition distribution and ordering (possibly adding new physical operators, e.g. `BroadcastExchangeExec` and `ShuffleExchangeExec` for distribution or `SortExec` for sorting)

Technically, `EnsureRequirements` is just a [Catalyst rule](#) for transforming [physical query plans](#), i.e. `Rule[SparkPlan]`.

`EnsureRequirements` is part of [preparations](#) batch of physical query plan rules and is executed when `QueryExecution` is requested for the [optimized physical query plan](#) (i.e. in `executedPlan` phase of a query execution).

`EnsureRequirements` takes a [SQLConf](#) when created.

```
val q = ??? // FIXME
val sparkPlan = q.queryExecution.sparkPlan

import org.apache.spark.sql.execution.exchange.EnsureRequirements
val plan = EnsureRequirements(spark.sessionState.conf).apply(sparkPlan)
```

createPartitioning Internal Method

Caution	FIXME
---------	-------

defaultNumPreShufflePartitions Internal Method

Caution	FIXME
---------	-------

Enforcing Partition Requirements (Distribution and Ordering) of Physical Operator

— ensureDistributionAndOrdering Internal Method

```
ensureDistributionAndOrdering(operator: SparkPlan): SparkPlan
```

Internally, `ensureDistributionAndOrdering` takes the following from the input physical operator :

- required partition requirements for the children
- required sort ordering per the required partition requirements per child
- child physical plans

Note	The number of requirements for partitions and their sort ordering has to match the number and the order of the child physical plans.
------	--

`ensureDistributionAndOrdering` matches the operator's required partition requirements of children (`requiredChildDistributions`) to the children's output partitioning and (in that order):

1. If the child satisfies the requested distribution, the child is left unchanged
2. For `BroadcastDistribution`, the child becomes the child of `BroadcastExchangeExec` unary operator for broadcast hash joins
3. Any other pair of child and distribution leads to `ShuffleExchangeExec` unary physical operator (with proper partitioning for distribution and with `spark.sql.shuffle.partitions` number of partitions, i.e. `200` by default)

Note	<code>ShuffleExchangeExec</code> can appear in the physical plan when the children's output partitioning cannot satisfy the physical operator's required child distribution.
------	--

If the input `operator` has multiple children and specifies child output distributions, then the children's output partitionings have to be compatible.

If the children's output partitionings are not all compatible, then...FIXME

`ensureDistributionAndOrdering` adds `ExchangeCoordinator` (only when adaptive query execution is enabled which is not by default).

Note	At this point in <code>ensureDistributionAndOrdering</code> the required child distributions are already handled.
------	---

`ensureDistributionAndOrdering` matches the operator's required sort ordering of children (`requiredChildOrderings`) to the children's output partitioning and if the orderings do not match, `SortExec` unary physical operator is created as a new child.

In the end, `ensureDistributionAndOrdering` sets the new children for the input `operator`.

Note

`ensureDistributionAndOrdering` is used exclusively when `EnsureRequirements` is **executed** (i.e. applied to a physical plan).

Adding ExchangeCoordinator (Adaptive Query Execution) — `withExchangeCoordinator` Internal Method

```
withExchangeCoordinator(
    children: Seq[SparkPlan],
    requiredChildDistributions: Seq[Distribution]): Seq[SparkPlan]
```

`withExchangeCoordinator` adds `ExchangeCoordinator` to `ShuffleExchangeExec` operators if adaptive query execution is enabled (per `spark.sql.adaptive.enabled` property) and partitioning scheme of the `shuffleExchangeExec` operators support `ExchangeCoordinator`.

Note

`spark.sql.adaptive.enabled` property is disabled by default.

Internally, `withExchangeCoordinator` checks if the input `children` operators support `ExchangeCoordinator` which is that either holds:

- If there is at least one `ShuffleExchangeExec` operator, all children are either `ShuffleExchangeExec` with `HashPartitioning` or their `output partitioning` is `HashPartitioning` (even inside `PartitioningCollection`)
- There are at least two `children` operators and the input `requiredChildDistributions` are all `ClusteredDistribution`

With `adaptive query execution` (i.e. when `spark.sql.adaptive.enabled` configuration property is `true`) and the `operator supports ExchangeCoordinator`, `withExchangeCoordinator` creates a `ExchangeCoordinator` and:

- For every `ShuffleExchangeExec`, registers the `ExchangeCoordinator`
- Creates `HashPartitioning partitioning scheme` with the `default number of partitions` to use when shuffling data for joins or aggregations (as `spark.sql.shuffle.partitions` which is `200` by default) and adds `ShuffleExchangeExec` to the final result (for the current physical operator)

Otherwise (when adaptive query execution is disabled or `children` do not support `ExchangeCoordinator`), `withExchangeCoordinator` returns the input `children` unchanged.

Note

`withExchangeCoordinator` is used exclusively for `enforcing partition requirements of a physical operator`.

reorderJoinPredicates Internal Method

```
reorderJoinPredicates(plan: SparkPlan): SparkPlan
```

reorderJoinPredicates ...FIXME

Note

reorderJoinPredicates is used when...FIXME

ExtractPythonUDFs Physical Query Optimization

`ExtractPythonUDFs` is a **physical query optimization** (aka *physical query preparation rule* or simply *preparation rule*) that `QueryExecution` uses to optimize the physical plan of a structured query by extracting Python UDFs from a physical query plan (excluding `FlatMapGroupsInPandasExec` operators that it simply skips over).

Technically, `ExtractPythonUDFs` is just a [Catalyst rule](#) for transforming physical query plans, i.e. `Rule[SparkPlan]`.

`ExtractPythonUDFs` is part of [preparations](#) batch of physical query plan rules and is executed when `QueryExecution` is requested for the [optimized physical query plan](#) (i.e. in `executedPlan` phase of a query execution).

Extracting Python UDFs from Physical Query Plan — `extract` Internal Method

```
extract(plan: SparkPlan): SparkPlan
```

```
extract ...FIXME
```

Note

`extract` is used exclusively when `ExtractPythonUDFs` is requested to [optimize a physical query plan](#).

trySplitFilter Internal Method

```
trySplitFilter(plan: SparkPlan): SparkPlan
```

```
trySplitFilter ...FIXME
```

Note

`trySplitFilter` is used exclusively when `ExtractPythonUDFs` is requested to [extract](#).

PlanSubqueries Physical Query Optimization

`PlanSubqueries` is a **physical query optimization** (aka *physical query preparation rule* or simply *preparation rule*) that [plans `ScalarSubquery` \(`SubqueryExpression`\) expressions](#) (as `ScalarSubquery ExecSubqueryExpression` expressions).

```

import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.execution.PlanSubqueries
val planSubqueries = PlanSubqueries(spark)

Seq(
  (0, 0),
  (1, 0),
  (2, 1)
).toDF("id", "gid").createOrReplaceTempView("t")

Seq(
  (0, 3),
  (1, 20)
).toDF("gid", "lvl").createOrReplaceTempView("v")

val sql = """
  select * from t where gid > (select max(gid) from v)
"""
val q = spark.sql(sql)

val sparkPlan = q.queryExecution.sparkPlan
scala> println(sparkPlan.numberedTreeString)
00 Project [_1#49 AS id#52, _2#50 AS gid#53]
01 +- Filter (_2#50 > scalar-subquery#128 [])
02   :   +- Aggregate [max(gid#61) AS max(gid)#130]
03   :     +- LocalRelation [gid#61]
04   +- LocalTableScan [_1#49, _2#50]

val optimizedPlan = planSubqueries(sparkPlan)
scala> println(optimizedPlan.numberedTreeString)
00 Project [_1#49 AS id#52, _2#50 AS gid#53]
01 +- Filter (_2#50 > Subquery subquery128)
02   :   +- Subquery subquery128
03   :     +- *(2) HashAggregate(keys=[], functions=[max(gid#61)], output=[max(gid)#130])
04   :       +- Exchange SinglePartition
05   :         +- *(1) HashAggregate(keys=[], functions=[partial_max(gid#61)], output=[max#134])
06   :           +- LocalTableScan [gid#61]
07   +- LocalTableScan [_1#49, _2#50]
```

`PlanSubqueries` is part of [preparations](#) batch of physical query plan rules and is executed when `QueryExecution` is requested for the [optimized physical query plan](#) (i.e. in **executedPlan** phase of a query execution).

Technically, `PlanSubqueries` is just a [Catalyst rule](#) for transforming [physical query plans](#), i.e. `Rule[SparkPlan]`.

Applying PlanSubqueries Rule to Physical Plan (Executing PlanSubqueries) — apply Method

```
apply(plan: SparkPlan): SparkPlan
```

Note `apply` is part of [Rule Contract](#) to apply a rule to a [TreeNode](#), e.g. [physical plan](#).

For every [ScalarSubquery \(SubqueryExpression\)](#) expression in the input [physical plan](#), `apply` does the following:

1. Builds the [optimized physical plan](#) (aka `executedPlan`) of the [subquery logical plan](#), i.e. creates a [QueryExecution](#) for the subquery logical plan and requests the optimized physical plan.
2. Plans the scalar subquery, i.e. creates a [ScalarSubquery \(ExecSubqueryExpression\)](#) expression with a new [SubqueryExec](#) physical operator (with the name **subquery[id]** and the optimized physical plan) and the [ExprId](#).

ReuseExchange Physical Query Optimization

`ReuseExchange` is a **physical query optimization** (aka *physical query preparation rule* or simply *preparation rule*) that `QueryExecution` uses to optimize the physical plan of a structured query by [FIXME](#).

Technically, `ReuseExchange` is just a [Catalyst rule](#) for transforming [physical query plans](#), i.e. `Rule[SparkPlan]`.

`ReuseExchange` is part of [preparations](#) batch of physical query plan rules and is executed when `QueryExecution` is requested for the [optimized physical query plan](#) (i.e. in **executedPlan** phase of a query execution).

apply Method

```
apply(plan: SparkPlan): SparkPlan
```

Note `apply` is part of [Rule Contract](#) to apply a rule to a [physical plan](#).

`apply` finds all [Exchange](#) unary operators and...[FIXME](#)

`apply` does nothing and simply returns the input physical `plan` if `spark.sql.exchange.reuse` internal configuration property is off (i.e. `false`).

Note `spark.sql.exchange.reuse` internal configuration property is on (i.e. `true`) by default.

ReuseSubquery Physical Query Optimization

`ReuseSubquery` is a **physical query optimization** (aka *physical query preparation rule* or simply *preparation rule*) that `QueryExecution` uses to optimize the physical plan of a structured query by [FIXME](#).

Technically, `ReuseSubquery` is just a [Catalyst rule](#) for transforming [physical query plans](#), i.e. `Rule[SparkPlan]`.

`ReuseSubquery` is part of [preparations](#) batch of physical query plan rules and is executed when `QueryExecution` is requested for the [optimized physical query plan](#) (i.e. in **executedPlan** phase of a query execution).

apply Method

```
apply(plan: SparkPlan): SparkPlan
```

Note	<code>apply</code> is part of Rule Contract to apply a rule to a physical plan .
------	--

`apply` ...[FIXME](#)

Encoder—Internal Row Converter

Encoder is the fundamental concept in the **serialization and deserialization (SerDe) framework** in Spark SQL 2.0. Spark SQL uses the SerDe framework for IO to make it efficient time- and space-wise.

Tip

Spark has borrowed the idea from the [Hive SerDe library](#) so it might be worthwhile to get familiar with Hive a little bit, too.

Encoders are modelled in Spark SQL 2.0 as `Encoder[T]` trait.

```
trait Encoder[T] extends Serializable {
  def schema: StructType
  def clsTag: ClassTag[T]
}
```

The type `T` stands for the type of records a `Encoder[T]` can deal with. An encoder of type `T`, i.e. `Encoder[T]`, is used to convert (*encode* and *decode*) any JVM object or primitive of type `T` (that could be your domain object) to and from Spark SQL's [InternalRow](#) which is the internal binary row format representation (using Catalyst expressions and code generation).

Note

`Encoder` is also called "*a container of serde expressions in Dataset*".

Note

The one and only implementation of the `Encoder` trait in Spark SQL 2 is [ExpressionEncoder](#).

Encoders are integral (and internal) part of any [Dataset\[T\]](#) (of records of type `T`) with a `Encoder[T]` that is used to serialize and deserialize the records of this dataset.

Note

`Dataset[T]` type is a Scala type constructor with the type parameter `T`. So is `Encoder[T]` that handles serialization and deserialization of `T` to the internal representation.

Encoders know the `schema` of the records. This is how they offer significantly faster serialization and deserialization (comparing to the default Java or Kryo serializers).

```
// The domain object for your records in a large dataset
case class Person(id: Long, name: String)

import org.apache.spark.sql.Encoders

scala> val personEncoder = Encoders.product[Person]
personEncoder: org.apache.spark.sql.Encoder[Person] = class[id[0]: bigint, name[0]: st
```

```

ring]

scala> personEncoder.schema
res0: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType, false
), StructField(name,StringType, true))

scala> personEncoder.clsTag
res1: scala.reflect.ClassTag[Person] = Person

import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder

scala> val personExprEncoder = personEncoder.asInstanceOf[ExpressionEncoder[Person]]
personExprEncoder: org.apache.spark.sql.catalyst.encoders.ExpressionEncoder[Person] =
class[id[0]: bigint, name[0]: string]

// ExpressionEncoders may or may not be flat
scala> personExprEncoder.flat
res2: Boolean = false

// The Serializer part of the encoder
scala> personExprEncoder.serializer
res3: Seq[org.apache.spark.sql.catalyst.expressions.Expression] = List(assertnonnull(i
nput[0, Person, true], top level non-flat input object).id AS id#0L, staticinvoke(class
org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnonnull(inpu
t[0, Person, true], top level non-flat input object).name, true) AS name#1)

// The Deserializer part of the encoder
scala> personExprEncoder.deserializer
res4: org.apache.spark.sql.catalyst.expressions.Expression = newInstance(class Person)

scala> personExprEncoder.namedExpressions
res5: Seq[org.apache.spark.sql.catalyst.expressions.NamedExpression] = List(assertnonnull(i
nput[0, Person, true], top level non-flat input object).id AS id#2L, staticinvoke(
class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnonnull(
input[0, Person, true], top level non-flat input object).name, true) AS name#3)

// A record in a Dataset[Person]
// A mere instance of Person case class
// There could be a thousand of Person in a large dataset
val jacek = Person(0, "Jacek")

// Serialize a record to the internal representation, i.e. InternalRow
scala> val row = personExprEncoder.toRow(jacek)
row: org.apache.spark.sql.catalyst.InternalRow = [0,0,1800000005,6b6563614a]

// Spark uses InternalRows internally for IO
// Let's deserialize it to a JVM object, i.e. a Scala object
import org.apache.spark.sql.catalyst.dsl.expressions._

// in spark-shell there are competing implicits
// That's why DslSymbol is used explicitly in the following line
scala> val attrs = Seq(DslSymbol('id).long, DslSymbol('name).string)
attrs: Seq[org.apache.spark.sql.catalyst.expressions.AttributeReference] = List(id#8L,

```

```

name#9)

scala> val jacekReborn = personExprEncoder.resolveAndBind(attrs).fromRow(row)
jacekReborn: Person = Person(0, Jacek)

// Are the jacek instances same?
scala> jacek == jacekReborn
res6: Boolean = true

```

You can [create custom encoders using static methods of `Encoders` object](#). Note however that encoders for common Scala types and their product types are already available in `implicits` object.

```

val spark = SparkSession.builder.getOrCreate()
import spark.implicits._

```

Tip	The default encoders are already imported in spark-shell .
-----	--

Encoders map columns (of your dataset) to fields (of your JVM object) by name. It is by Encoders that you can bridge JVM objects to data sources (CSV, JDBC, Parquet, Avro, JSON, Cassandra, Elasticsearch, memsql) and vice versa.

Note	In Spark SQL 2.0 <code>DataFrame</code> type is a mere type alias for <code>Dataset[Row]</code> with RowEncoder being the encoder.
------	--

Creating Custom Encoders (Encoders object)

`Encoders` factory object defines methods to create `Encoder` instances.

Import `org.apache.spark.sql` package to have access to the `Encoders` factory object.

```

import org.apache.spark.sql.Encoders

scala> Encoders.LONG
res1: org.apache.spark.sql.Encoder[Long] = class[value[0]: bigint]

```

You can find methods to create encoders for Java's object types, e.g. `Boolean`, `Integer`, `Long`, `Double`, `String`, `java.sql.Timestamp` or `Byte` array, that could be composed to create more advanced encoders for Java bean classes (using `bean` method).

```
import org.apache.spark.sql.Encoders

scala> Encoders.STRING
res2: org.apache.spark.sql.Encoder[String] = class[value[0]: string]
```

You can also create encoders based on Kryo or Java serializers.

```
import org.apache.spark.sql.Encoders

case class Person(id: Int, name: String, speaksPolish: Boolean)

scala> Encoders.kryo[Person]
res3: org.apache.spark.sql.Encoder[Person] = class[value[0]: binary]

scala> Encoders.javaSerialization[Person]
res5: org.apache.spark.sql.Encoder[Person] = class[value[0]: binary]
```

You can create encoders for Scala's tuples and case classes, `Int`, `Long`, `Double`, etc.

```
import org.apache.spark.sql.Encoders

scala> Encoders.tuple(Encoders.scalaLong, Encoders.STRING, Encoders.scalaBoolean)
res9: org.apache.spark.sql.Encoder[(Long, String, Boolean)] = class[_1[0]: bigint, _2[0]
]: string, _3[0]: boolean]
```

Further Reading and Watching

- (video) [Modern Spark DataFrame and Dataset \(Intermediate Tutorial\)](#) by Adam Breindel from Databricks.

Encoders Factory Object

`Encoders` is a factory object that...FIXME

Creating Encoder Using Kryo — `kryo` Method

```
kryo[T: ClassTag]: Encoder[T]
```

`kryo` simply [creates an encoder](#) that serializes objects of type `T` using Kryo (i.e. the `useKryo` flag is enabled).

Note

`kryo` is used when...FIXME

Creating Encoder Using Java Serialization — `javaSerialization` Method

```
javaSerialization[T: ClassTag]: Encoder[T]
```

`javaSerialization` simply [creates an encoder](#) that serializes objects of type `T` using the generic Java serialization (i.e. the `useKryo` flag is disabled).

Note

`javaSerialization` is used when...FIXME

Creating Generic Encoder — `genericSerializer` Internal Method

```
genericSerializer[T: ClassTag](useKryo: Boolean): Encoder[T]
```

`genericSerializer` ...FIXME

Note

`genericSerializer` is used when `Encoders` is requested for a generic encoder using [Kryo](#) and [Java Serialization](#).

ExpressionEncoder — Expression-Based Encoder

`ExpressionEncoder[T]` is a generic [Encoder](#) of JVM objects of the type `T` to and from [internal binary rows](#).

`ExpressionEncoder[T]` uses [expressions](#) for a [serializer](#) and a [deserializer](#).

Note	<code>ExpressionEncoder</code> is the only supported implementation of <code>Encoder</code> which is explicitly enforced when <code>dataset</code> is created (even though <code>Dataset</code> data structure accepts a <i>bare</i> <code>Encoder[T]</code>).
------	---

```

import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val stringEncoder = ExpressionEncoder[String]
scala> val row = stringEncoder.toRow("hello world")
row: org.apache.spark.sql.catalyst.InternalRow = [0,100000000b,6f77206f6c6c6568,646c72
]

import org.apache.spark.sql.catalyst.expressions.UnsafeRow
scala> val unsafeRow = row match { case ur: UnsafeRow => ur }
unsafeRow: org.apache.spark.sql.catalyst.expressions.UnsafeRow = [0,100000000b,6f77206
f6c6c6568,646c72]

```

`ExpressionEncoder` uses [serializer expressions](#) to encode (aka *serialize*) a JVM object of type `T` to an [internal binary row format](#) (i.e. `InternalRow`).

Note	It is assumed that all serializer expressions contain at least one and the same BoundReference .
------	--

`ExpressionEncoder` uses a [deserializer expression](#) to decode (aka *deserialize*) a JVM object of type `T` from [internal binary row format](#).

`ExpressionEncoder` is [flat](#) when [serializer](#) uses a single expression (which also means that the objects of a type `T` are not created using constructor parameters only like `Product` or `DefinedByConstructorParams` types).

Internally, a `ExpressionEncoder` creates a [UnsafeProjection](#) (for the input serializer), a `InternalRow` (of size `1`), and a safe `Projection` (for the input deserializer). They are all internal lazy attributes of the encoder.

Table 1. ExpressionEncoder's (Lazily-Initialized) Internal Properties

Property	Description
<code>constructProjection</code>	<code>Projection</code> generated for the <code>deserializer</code> expression Used exclusively when <code>ExpressionEncoder</code> is requested for a <code>JVM object from a Spark SQL row</code> (i.e. <code>InternalRow</code>).
<code>extractProjection</code>	<code>UnsafeProjection</code> generated for the <code>serializer</code> expressions Used exclusively when <code>ExpressionEncoder</code> is requested for an <code>encoded version of a JVM object as a Spark SQL row</code> (i.e. <code>InternalRow</code>).
<code>inputRow</code>	<code>GenericInternalRow</code> (with the underlying storage array) of size 1 (i.e. it can only store a single JVM object of any type). Used...FIXME
Note	Encoders object contains the default <code>ExpressionEncoders</code> for Scala and Java primitive types, e.g. <code>boolean</code> , <code>long</code> , <code>String</code> , <code>java.sql.Date</code> , <code>java.sql.Timestamp</code> , <code>Array[Byte]</code> .

Creating ExpressionEncoder— apply Method

```
apply[T : TypeTag](): ExpressionEncoder[T]
```

Caution	FIXME
---------	-------

Creating ExpressionEncoder Instance

`ExpressionEncoder` takes the following when created:

- Schema
- Flag whether `ExpressionEncoder` is flat or not
- Serializer `expressions` (to convert objects of type `T` to internal rows)
- Deserializer `expression` (to convert internal rows to objects of type `T`)
- Scala's `ClassTag` for the JVM type `T`

Creating Deserialize Expression — `ScalaReflection.deserializerFor` Method

```
deserializerFor[T: TypeTag]: Expression
```

`deserializerFor` creates an expression to deserialize from internal binary row format to a Scala object of type `T`.

```
import org.apache.spark.sql.catalystScalaReflection.deserializerFor
val timestampDeExpr = deserializerFor[java.sql.Timestamp]
scala> println(timestampDeExpr.numberedTreeString)
00 staticinvoke(class org.apache.spark.sql.catalyst.util.DateTimeUtils$, ObjectType(clazz java.sql.Timestamp), toJavaTimestamp, upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - root class: "java.sql.Timestamp"), true)
01 +- upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - root class: "java.sql.Timestamp")
02   +- getcolumnbyordinal(0, TimestampType)

val tuple2DeExpr = deserializerFor[(java.sql.Timestamp, Double)]
scala> println(tuple2DeExpr.numberedTreeString)
00 newInstance(class scala.Tuple2)
01 :- staticinvoke(class org.apache.spark.sql.catalyst.util.DateTimeUtils$, ObjectType(class java.sql.Timestamp), toJavaTimestamp, upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - field (class: "java.sql.Timestamp", name: "_1"), - root class: "scala.Tuple2"), true)
02 : +- upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - field (class: "java.sql.Timestamp", name: "_1"), - root class: "scala.Tuple2")
03 :   +- getcolumnbyordinal(0, TimestampType)
04 +- upcast(getcolumnbyordinal(1, DoubleType), DoubleType, - field (class: "scala.Double", name: "_2"), - root class: "scala.Tuple2")
05   +- getcolumnbyordinal(1, DoubleType)
```

Internally, `deserializerFor` calls the recursive internal variant of `deserializerFor` with a single-element walked type path with `- root class: "[clsName]"`

Tip	Read up on Scala's <code>TypeTags</code> in TypeTags and Manifests .
-----	--

Note	<code>deserializerFor</code> is used exclusively when <code>ExpressionEncoder</code> is created for a Scala type <code>T</code> .
------	---

Recursive Internal `deserializerFor` Method

```
deserializerFor(
  tpe: `Type`,
  path: Option[Expression],
  walkedTypePath: Seq[String]): Expression
```

Table 2. JVM Types and Deserialize Expressions (in evaluation order)

JVM Type (Scala or Java)	Deserialize Expressions
<code>Option[T]</code>	
<code>java.lang.Integer</code>	
<code>java.lang.Long</code>	
<code>java.lang.Double</code>	
<code>java.lang.Float</code>	
<code>java.lang.Short</code>	
<code>java.lang.Byte</code>	
<code>java.lang.Boolean</code>	
<code>java.sql.Date</code>	
<code>java.sql.Timestamp</code>	
<code>java.lang.String</code>	
<code>java.math.BigDecimal</code>	
<code>scala.BigDecimal</code>	
<code>java.math.BigInteger</code>	
<code>scala.math.BigInt</code>	
<code>Array[T]</code>	
<code>Seq[T]</code>	
<code>Map[K, V]</code>	
<code>SQLUserDefinedType</code>	
User Defined Types (UDTs)	
Product (including <code>Tuple</code>) or DefinedByConstructorParams	

Creating Serialize Expression — `ScalaReflection.serializerFor` Method

```
serializerFor[T: TypeTag](inputObject: Expression): CreateNamedStruct
```

`serializerFor` creates a [CreateNamedStruct](#) expression to serialize a Scala object of type `T` to [internal binary row format](#).

```
import org.apache.spark.sql.catalystScalaReflection.serializerFor

import org.apache.spark.sql.catalyst.expressions.BoundReference
import org.apache.spark.sql.types.TimestampType
val boundRef = BoundReference(ordinal = 0, dataType = TimestampType, nullable = true)

val timestampSerExpr = serializerFor[java.sql.Timestamp](boundRef)
scala> println(timestampSerExpr.numberedTreeString)
00 named_struct(value, input[0, timestamp, true])
01 :- value
02 +- input[0, timestamp, true]
```

Internally, `serializerFor` calls the recursive internal variant of `serializerFor` with a single-element walked type path with `- root class: "[clsName]"` and *pattern match* on the result [expression](#).

Caution	FIXME the pattern match part
---------	------------------------------

Tip	Read up on Scala's <code>TypeTags</code> in TypeTags and Manifests .
-----	--

Note	<code>serializerFor</code> is used exclusively when <code>ExpressionEncoder</code> is created for a Scala type <code>T</code> .
------	---

Recursive Internal `serializerFor` Method

```
serializerFor(
  inputObject: Expression,
  tpe: `Type`,
  walkedTypePath: Seq[String],
  seenTypeSet: Set[`Type`] = Set.empty): Expression
```

`serializerFor` creates an [expression](#) for serializing an object of type `T` to an internal row.

Caution	FIXME
---------	-------

Encoding JVM Object to Internal Binary Row Format — `toRow` Method

```
toRow(t: T): InternalRow
```

`toRow` encodes (aka *serializes*) a JVM object `t` as an [internal binary row](#).

Internally, `toRow` sets the only JVM object to be `t` in `inputRow` and converts the `inputRow` to a [unsafe binary row](#) (using `extractProjection`).

In case of any exception while serializing, `toRow` reports a `RuntimeException`:

```
Error while encoding: [initial exception]
[multi-line serializer]
```

Note

`toRow` is *mostly* used when `SparkSession` is requested for:

- [Dataset from a local dataset](#)
- [DataFrame from RDD\[Row\]](#)

Decoding JVM Object From Internal Binary Row Format — `fromRow` Method

```
fromRow(row: InternalRow): T
```

`fromRow` decodes (aka *deserializes*) a JVM object from a `row` [InternalRow](#) (with the required values only).

Internally, `fromRow` uses `constructProjection` with `row` and gets the 0th element of type `ObjectType` that is then cast to the output type `T`.

In case of any exception while deserializing, `fromRow` reports a `RuntimeException`:

```
Error while decoding: [initial exception]
[deserializer]
```

Note

`fromRow` is used for:

- `Dataset` operators, i.e. `head`, `collect`, `collectAsList`, `toLocalIterator`
- Structured Streaming's `ForeachSink`

Creating ExpressionEncoder For Tuple — `tuple` Method

```

tuple(encoders: Seq[ExpressionEncoder[_]]): ExpressionEncoder[_]
tuple[T](e: ExpressionEncoder[T]): ExpressionEncoder[Tuple1[T]]
tuple[T1, T2](
  e1: ExpressionEncoder[T1],
  e2: ExpressionEncoder[T2]): ExpressionEncoder[(T1, T2)]
tuple[T1, T2, T3](
  e1: ExpressionEncoder[T1],
  e2: ExpressionEncoder[T2],
  e3: ExpressionEncoder[T3]): ExpressionEncoder[(T1, T2, T3)]
tuple[T1, T2, T3, T4](
  e1: ExpressionEncoder[T1],
  e2: ExpressionEncoder[T2],
  e3: ExpressionEncoder[T3],
  e4: ExpressionEncoder[T4]): ExpressionEncoder[(T1, T2, T3, T4)]
tuple[T1, T2, T3, T4, T5](
  e1: ExpressionEncoder[T1],
  e2: ExpressionEncoder[T2],
  e3: ExpressionEncoder[T3],
  e4: ExpressionEncoder[T4],
  e5: ExpressionEncoder[T5]): ExpressionEncoder[(T1, T2, T3, T4, T5)]

```

`tuple ...FIXME`

Note

`tuple` is used when...FIXME

resolveAndBind Method

```

resolveAndBind(
  attrs: Seq[Attribute] = schema.toAttributes,
  analyzer: Analyzer = SimpleAnalyzer): ExpressionEncoder[T]

```

`resolveAndBind ...FIXME`

```
// A very common use case
case class Person(id: Long, name: String)
import org.apache.spark.sql.Encoders
val schema = Encoders.product[Person].schema

import org.apache.spark.sql.catalyst.encoders.{RowEncoder, ExpressionEncoder}
import org.apache.spark.sql.Row
val encoder: ExpressionEncoder[Row] = RowEncoder.apply(schema).resolveAndBind()

import org.apache.spark.sql.catalyst.InternalRow
val row = InternalRow(1, "Jacek")

val deserializer = encoder.deserializer

scala> deserializer.eval(row)
java.lang.UnsupportedOperationException: Only code-generated evaluation is supported
  at org.apache.spark.sql.catalyst.expressions.objects.CreateExternalRow.eval(objects.
scala:1105)
... 54 elided

import org.apache.spark.sql.catalyst.expressions.codegen.CodegenContext
val ctx = new CodegenContext
val code = deserializer.genCode(ctx).code
```

Note

`resolveAndBind` is used when:

- `InternalRowDataWriterFactory` is requested to [create a DataWriter](#)
- `Dataset` is requested for the [deserializer expression](#) (to convert internal rows to objects of type `T`)
- `TypedAggregateExpression` is [created](#)
- `JdbcUtils` is requested to [resultSetToRows](#)
- Spark Structured Streaming's `FlatMapGroupsWithStateExec` physical operator is requested for the state deserializer (i.e. `stateDeserializer`)
- Spark Structured Streaming's `ForeachSink` is requested to add a streaming batch (i.e. `addBatch`)

RowEncoder — Encoder for DataFrames

`RowEncoder` is part of the [Encoder framework](#) and acts as the encoder for [DataFrames](#), i.e. `Dataset[Row]` — [Datasets of Rows](#).

Note

`DataFrame` type is a mere type alias for `Dataset[Row]` that expects a `Encoder[Row]` available in scope which is indeed `RowEncoder` itself.

`RowEncoder` is an `object` in Scala with [apply](#) and other factory methods.

`RowEncoder` can create `ExpressionEncoder[Row]` from a [schema](#) (using [apply method](#)).

```
import org.apache.spark.sql.types._
val schema = StructType(
  StructField("id", LongType, nullable = false) ::
  StructField("name", StringType, nullable = false) :: Nil)

import org.apache.spark.sql.catalyst.encoders.RowEncoder
scala> val encoder = RowEncoder(schema)
encoder: org.apache.spark.sql.catalyst.encoders.ExpressionEncoder[org.apache.spark.sql.
Row] = class[id[0]: bigint, name[0]: string]

// RowEncoder is never flat
scala> encoder.flat
res0: Boolean = false
```

`RowEncoder` object belongs to `org.apache.spark.sql.catalyst.encoders` package.

Creating ExpressionEncoder For Row Type — [apply](#) method

```
apply(schema: StructType): ExpressionEncoder[Row]
```

`apply` builds [ExpressionEncoder](#) of `Row`, i.e. `ExpressionEncoder[Row]`, from the input `StructType` (as `schema`).

Internally, `apply` creates a [BoundReference](#) for the `Row` type and returns a `ExpressionEncoder[Row]` for the input `schema`, a `CreateNamedStruct` serializer (using `serializerFor` [internal method](#)), a deserializer for the schema, and the `Row` type.

serializerFor Internal Method

```
serializerFor(inputObject: Expression, inputType: DataType): Expression
```

`serializerFor` creates an `Expression` that is assumed to be `CreateNamedStruct`.

`serializerFor` takes the input `inputType` and:

1. Returns the input `inputObject` as is for native types, i.e. `NullType`, `BooleanType`, `ByteType`, `ShortType`, `IntegerType`, `LongType`, `FloatType`, `DoubleType`, `BinaryType`, `CalendarIntervalType`.

Caution	FIXME What does being native type mean?
---------	---

2. For `UserDefinedType`s, it takes the UDT class from the `SQLUserDefinedType` annotation or `UDTRegistration` object and returns an expression with `Invoke` to call `serialize` method on a `NewInstance` of the UDT class.
3. For `TimestampType`, it returns an expression with a `StaticInvoke` to call `fromJavaTimestamp` on `DateTimeUtils` class.
4. ...FIXME

Caution	FIXME Describe me.
---------	--------------------

LocalDateTimeEncoder — Custom ExpressionEncoder for java.time.LocalDateTime

Spark SQL does not support `java.time.LocalDateTime` values in a `Dataset`.

```

import java.time.LocalDateTime

val data = Seq((0, LocalDateTime.now))
scala> val times = data.toDF("time")
java.lang.UnsupportedOperationException: No Encoder found for java.time.LocalDateTime
- field (class: "java.time.LocalDateTime", name: "_2")
- root class: "scala.Tuple2"
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:643)
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:445)
  at scala.reflect.internal.tpe.TypeConstraints$UndoLog.undo(TypeConstraints.scala:56)
  at org.apache.spark.sql.catalystScalaReflection$class.cleanUpReflectionObjects(ScalaReflection.scala:824)
  at org.apache.spark.sql.catalystScalaReflection$.cleanUpReflectionObjects(ScalaReflection.scala:39)
  at org.apache.spark.sql.catalystScalaReflection$.org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor(ScalaReflection.scala:445)
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1$$anonfun$8.apply(ScalaReflection.scala:637)
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1$$anonfun$8.apply(ScalaReflection.scala:625)
  at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:241)
  at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:241)
  at scala.collection.immutable.List.foreach(List.scala:381)
  at scala.collection.TraversableLike$class.flatMap(TraversableLike.scala:241)
  at scala.collection.immutable.List.flatMap(List.scala:344)
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:625)
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:445)
  at scala.reflect.internal.tpe.TypeConstraints$UndoLog.undo(TypeConstraints.scala:56)
  at org.apache.spark.sql.catalystScalaReflection$class.cleanUpReflectionObjects(ScalaReflection.scala:824)
  at org.apache.spark.sql.catalystScalaReflection$.cleanUpReflectionObjects(ScalaReflection.scala:39)
  at org.apache.spark.sql.catalystScalaReflection$.org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor(ScalaReflection.scala:445)
  at org.apache.spark.sql.catalystScalaReflection$.serializerFor(ScalaReflection.scala:434)
  at org.apache.spark.sql.catalyst.encoders.ExpressionEncoder$.apply(ExpressionEncoder.scala:71)
  at org.apache.spark.sql.Encoders$.product(Encoders.scala:275)
  at org.apache.spark.sql.LowPrioritySQLImplicits$class.newProductEncoder(SQLImplicits.scala:248)
  at org.apache.spark.sql.SQLImplicits.newProductEncoder(SQLImplicits.scala:34)
  ... 50 elided

```

As it is clearly said in the exception, the root cause is no [Encoder](#) found for `java.time.LocalDateTime` (as there is not one available in Spark SQL).

You could define one using [ExpressionEncoder](#), but that does not seem to work either.

```
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
scala> ExpressionEncoder[java.time.LocalDateTime]
java.lang.UnsupportedOperationException: No Encoder found for java.time.LocalDateTime
- root class: "java.time.LocalDateTime"
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:643)
  at org.apache.spark.sql.catalystScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:445)
  at scala.reflect.internal.tpe.TypeConstraints$UndoLog.undo(TypeConstraints.scala:56)
  at org.apache.spark.sql.catalystScalaReflection$class.cleanupReflectionObjects(ScalaReflection.scala:824)
  at org.apache.spark.sql.catalystScalaReflection$.cleanupReflectionObjects(ScalaReflection.scala:39)
  at org.apache.spark.sql.catalystScalaReflection$.org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor(ScalaReflection.scala:445)
  at org.apache.spark.sql.catalystScalaReflection$.serializerFor(ScalaReflection.scala:434)
  at org.apache.spark.sql.catalyst.encoders.ExpressionEncoder$.apply(ExpressionEncoder.scala:71)
  ... 50 elided
```

The simplest solution is to transform the `Dataset` with `java.time.LocalDateTime` to a supported type that Spark SQL offers an encoder for.

A much better solution would be to provide a custom `Encoder` that would expand the types supported in Spark SQL.

`LocalDateTimeEncoder` is an *attempt* to develop a custom [ExpressionEncoder](#) for Java's `java.time.LocalDateTime` so you don't have to map values to another supported type.

`public final class LocalDateTime`

A date-time without a time-zone in the ISO-8601 calendar system, such as `2007-12-03T10:15:30`.

`LocalDateTime` is an immutable date-time object that represents a date-time, often viewed as year-month-day-hour-minute-second.

```
// A very fresh attempt at creating an Encoder for java.time.LocalDateTime

// See ExpressionEncoder.apply
import org.apache.spark.sql.catalyst.expressions.BoundReference
import org.apache.spark.sql.catalystScalaReflection
import java.time.LocalDateTime
```

```

val inputObject = BoundReference(0, ScalaReflection.dataTypeFor[LocalDateTime], nullable = true)

// ScalaReflection.serializerFor[LocalDateTime](inputObject)
import org.apache.spark.sql.catalyst.expressions.{CreateNamedStruct, Literal}
import org.apache.spark.sql.catalyst.expressions.objects.StaticInvoke
import org.apache.spark.sql.catalyst.util.DateTimeUtils
import org.apache.spark.sql.types.DateType
// Simply invokes DateTimeUtils.fromJavaDate
// fromJavaDate(date: Date): Int
// serializing a Date to an Int

// Util object to do conversion (serialization)
object LocalDateTimeUtils {
    import java.time.LocalDateTime
    def fromLocalDateTime(date: LocalDateTime): Long = date.toEpochSecond(java.time.ZoneOffset.UTC)
}

val other = StaticInvoke(
    LocalDateTimeUtils.getClass,
    DateType,
    "fromLocalDateTime",
    inputObject :: Nil,
    returnNullable = false)
val serializer = CreateNamedStruct(Literal("value") :: other :: Nil)
val schema = serializer.dataType

// ScalaReflection.deserializerFor[T]
// FIXME Create it as for ScalaReflection.serializerFor above
val deserializer = serializer // FIXME

import scala.reflect.ClassTag
import scala.reflect.runtime.universe.{typeTag, TypeTag}
val mirror = ScalaReflection.mirror
val tpe = typeTag[java.time.LocalDateTime].in(mirror).tpe
val cls = mirror.runtimeClass(tpe)

import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val localDateTimeEncoder = new ExpressionEncoder[java.time.LocalDateTime](
    schema,
    flat = true,
    serializer.flatten,
    deserializer,
    ClassTag[java.time.LocalDateTime](cls))

import org.apache.spark.sql.Encoder
implicit val encLocalDateTime: Encoder[java.time.LocalDateTime] = localDateTimeEncoder

// DEMO
val data = Seq(LocalDateTime.now)
val times = spark.createDataset(data) // (encLocalDateTime)

```

```
// $ SPARK_SUBMIT_OPTS="-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005" ./bin/spark-shell --conf spark.rpc.askTimeout=5m

import java.time.LocalDateTime
import org.apache.spark.sql.Encoder
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder

import org.apache.spark.sql.types._
val schema = StructType(
  $"year".int :: $"month".int :: $"day".int :: Nil)
import org.apache.spark.sql.catalyst.expressions.Expression
import org.apache.spark.sql.catalyst.expressions.objects.StaticInvoke

import org.apache.spark.sql.types.ObjectType
import org.apache.spark.sql.catalyst.expressions.BoundReference
val clazz = classOf[java.time.LocalDateTime]
val inputObject = BoundReference(0, ObjectType(clazz), nullable = true)
val nullSafeInput = inputObject

import org.apache.spark.sql.types.TimestampType
val staticInvoke = StaticInvoke(
  classOf[java.time.LocalDateTime],
  TimestampType,
  "parse",
  inputObject :: Nil)

// Based on UDTRegistration
val clazz = classOf[java.time.LocalDateTime]
import org.apache.spark.sql.catalyst.expressions.objects.NewInstance
val obj = NewInstance(
  cls = clazz,
  arguments = Nil,
  dataType = ObjectType(clazz))
import org.apache.spark.sql.catalyst.expressions.objects.Invoke

// the following would be nice to have
// FIXME How to bind them all up into one BoundReference?
import org.apache.spark.sql.types.IntegerType
val yearRef = BoundReference(0, IntegerType, nullable = true)
val monthRef = BoundReference(1, IntegerType, nullable = true)
val dayOfMonthRef = BoundReference(2, IntegerType, nullable = true)
val hourRef = BoundReference(3, IntegerType, nullable = true)
val minuteRef = BoundReference(4, IntegerType, nullable = true)

import org.apache.spark.sql.types.ArrayType
val inputObject = BoundReference(0, ArrayType(IntegerType), nullable = true)

def invoke(inputObject: Expression, fieldName: String) = Invoke(
  targetObject = inputObject,
  functionName = fieldName,
  dataType = IntegerType)
```

```

import org.apache.spark.sql.catalyst.expressions.CreateNamedStruct
import org.apache.spark.sql.catalyst.expressions.Literal
import org.apache.spark.sql.catalyst.expressions.GetArrayItem
val year = GetArrayItem(inputObject, Literal(0))
val month = GetArrayItem(inputObject, Literal(1))
val day = GetArrayItem(inputObject, Literal(2))
val hour = GetArrayItem(inputObject, Literal(3))
val minute = GetArrayItem(inputObject, Literal(4))

// turn LocalDateTime into InternalRow
// by saving LocalDateTime in parts
val serializer = CreateNamedStruct(
  Literal("year") :: year :::
  Literal("month") :: month :::
  Literal("day") :: day :::
  Literal("hour") :: hour :::
  Literal("minute") :: minute :::
  Nil)

import org.apache.spark.sql.catalyst.expressions.objects.StaticInvoke
import org.apache.spark.sql.catalyst.util.DateTimeUtils
val getPath: Expression = Literal("value")
val deserializer: Expression =
  StaticInvoke(
    DateTimeUtils.getClass,
    ObjectType(classOf[java.time.LocalDateTime]),
    "toJavaTimestamp",
    getPath :: Nil)

// we ask serializer about the schema
val schema: StructType = serializer.dataType

import scala.reflect._
implicit def scalaLocalDateTime: Encoder[java.time.LocalDateTime] =
  new ExpressionEncoder[java.time.LocalDateTime](
    schema,
    flat = false, // serializer.size == 1
    serializer.flatten,
    deserializer,
    classTag[java.time.LocalDateTime])

// the above leads to the following exception
// Add log4j.logger.org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator=DE
BUG to see the code
scala> scalaLocalDateTime.asInstanceOf[ExpressionEncoder[LocalDateTime]].toRow(LocalDateTime.now)
java.lang.RuntimeException: Error while encoding: java.lang.ClassCastException: java.time.LocalDateTime cannot be cast to org.apache.spark.sql.catalyst.util.ArrayData
input[0, array<int>, true][0] AS year#0
input[0, array<int>, true][1] AS month#1
input[0, array<int>, true][2] AS day#2
input[0, array<int>, true][3] AS hour#3
input[0, array<int>, true][4] AS minute#4

```

```

at org.apache.spark.sql.catalyst.encoders.ExpressionEncoder.toRow(ExpressionEncoder.
scala:291)
... 52 elided
Caused by: java.lang.ClassCastException: java.time.LocalDateTime cannot be cast to org
.apache.spark.sql.catalyst.util.ArrayData
at org.apache.spark.sql.catalyst.expressions.BaseGenericInternalRow$class.getArray(r
ows.scala:48)
at org.apache.spark.sql.catalyst.expressions.GenericInternalRow.getArray(rows.scala:
194)
at org.apache.spark.sql.catalyst.expressions.GeneratedClass$SpecificUnsafeProjection
.apply_0$(Unknown Source)
at org.apache.spark.sql.catalyst.expressions.GeneratedClass$SpecificUnsafeProjection
.apply(Unknown Source)
at org.apache.spark.sql.catalyst.encoders.ExpressionEncoder.toRow(ExpressionEncoder.
scala:288)
... 52 more

// and so the following won't work either
val times = Seq(LocalDateTime.now).toDF("time")

```

Open Questions

1. `ScalaReflection.serializerFor` passes `ObjectType` objects through
2. `ScalaReflection.serializerFor` uses `StaticInvoke` for `java.sql.Timestamp` and `java.sql.Date`.

```

case t if t <:< localTypeOf[java.sql.Timestamp] =>
  StaticInvoke(
    DateTimeUtils.getClass,
    TimestampType,
    "fromJavaTimestamp",
    inputObject :: Nil)

case t if t <:< localTypeOf[java.sql.Date] =>
  StaticInvoke(
    DateTimeUtils.getClass,
    DateType,
    "fromJavaDate",
    inputObject :: Nil)

```

3. How could `SQLUserDefinedType` and `UDTRegistration` help here?

ShuffledRowRDD

`ShuffledRowRDD` is an `RDD` of [internal binary rows](#) (i.e. `RDD[InternalRow]`) that is [created](#) when:

- `ShuffleExchangeExec` physical operator is requested to [preparePostShuffleRDD](#)
- `CollectLimitExec` and `TakeOrderedAndProjectExec` physical operators are requested to `doExecute`

`ShuffledRowRDD` takes the following to be created:

- `ShuffleDependency[Int, InternalRow, InternalRow]`
- Optional **partition start indices** (`Option[Array[Int]]`, default: `None`)

Note	The dependency property is mutable so it can be cleared .
------	---

`ShuffledRowRDD` takes an optional [partition start indices](#) that is the number of post-shuffle partitions. When not specified (when executing `CollectLimitExec` and `TakeOrderedAndProjectExec` physical operators), the number of post-shuffle partitions is managed by the [Partitioner](#) of the input `ShuffleDependency`. Otherwise, when specified (when `ExchangeCoordinator` is requested to [doEstimationIfNecessary](#)), `ShuffledRowRDD` ...
FIXME

Note	Post-shuffle partition is...FIXME
------	--

Note	<code>ShuffledRowRDD</code> looks like <code>ShuffledRDD</code> , and the difference is in the type of the values to process, i.e. <code>InternalRow</code> and <code>(k, v)</code> key-value pairs, respectively.
------	--

Table 1. ShuffledRowRDD and RDD Contract

Name	Description
<code>getDependencies</code>	A single-element collection with <code>ShuffleDependency[Int, InternalRow, InternalRow]</code> .
<code>partitioner</code>	CoalescedPartitioner (with the Partitioner of the dependency)
<code>getPreferredLocations</code>	
<code>compute</code>	

numPreShufflePartitions Property

Caution

FIXME

Computing Partition (in TaskContext) — `compute` Method

```
compute(split: Partition, context: TaskContext): Iterator[InternalRow]
```

Note

`compute` is part of Spark Core's `RDD` Contract to compute a partition (in a `TaskContext`).

Internally, `compute` makes sure that the input `split` is a `ShuffledRowRDDPartition`. It then requests `ShuffleManager` for a `ShuffleReader` to read `InternalRow`s for the `split`.

Note

`compute` uses `SparkEnv` to access the current `ShuffleManager`.

Note

`compute` uses `ShuffleHandle` (of `ShuffleDependency` dependency) and the pre-shuffle start and end partition offsets.

Getting Placement Preferences of Partition — `getPreferredLocations` Method

```
getPreferredLocations(partition: Partition): Seq[String]
```

Note

`getPreferredLocations` is part of `RDD contract` to specify placement preferences (aka *preferred task locations*), i.e. where tasks should be executed to be as close to the data as possible.

Internally, `getPreferredLocations` requests `MapOutputTrackerMaster` for the preferred locations of the input `partition` (for the single `ShuffleDependency`).

Note

`getPreferredLocations` uses `SparkEnv` to access the current `MapOutputTrackerMaster` (which runs on the driver).

CoalescedPartitioner

Caution

FIXME

ShuffledRowRDDPartition

Caution

FIXME

Clearing Dependencies — `clearDependencies` Method

```
clearDependencies(): Unit
```

Note

`clearDependencies` is part of the RDD Contract to clear dependencies of the RDD (to enable the parent RDDs to be garbage collected).

`clearDependencies` simply requests the parent RDD to `clearDependencies` followed by clear the given [dependency](#) (i.e. set to `null`).

SQL Tab — Monitoring Structured Queries in web UI

SQL tab in [web UI](#) shows [SQLMetrics](#) per [physical operator](#) in a structured query physical plan.

You can access the SQL tab under `/SQL` URL, e.g. <http://localhost:4040/SQL/>.

By default, it displays [all SQL query executions](#). However, after a query has been selected, the SQL tab [displays the details for the structured query execution](#).

AllExecutionsPage

`AllExecutionsPage` displays all SQL query executions in a Spark application per state sorted by their submission time reversed.



Figure 1. SQL Tab in web UI (AllExecutionsPage)

Internally, the page requests [SQLListener](#) for query executions in running, completed, and failed states (the states correspond to the respective tables on the page).

ExecutionPage — Details for Query

`ExecutionPage` shows details for structured query execution by `id`.

Note	The <code>id</code> request parameter is mandatory.
------	---

`ExecutionPage` displays a summary with **Submitted Time**, **Duration**, the clickable identifiers of the **Running Jobs**, **Succeeded Jobs**, and **Failed Jobs**.

It also display a visualization (using [accumulator updates](#) and the [SparkPlanGraph](#) for the query) with the expandable **Details** section (that corresponds to `SQLExecutionUIData.physicalPlanDescription`).



Figure 2. Details for Query in web UI

If there is no information to display for a given query `id`, you should see the following page.



Figure 3. No Details for SQL Query

Internally, it uses [SQLListener](#) exclusively to get the SQL query execution metrics. It requests [SQLListener](#) for [SQL execution data](#) to display for the `id` request parameter.

Creating SQLTab Instance

`SQLTab` is created when `SharedState` is or at the first `SparkListenerSQLExecutionStart` event when `Spark History Server` is used.



Figure 4. Creating SQLTab Instance

Note

`SharedState` represents the shared state across `SparkSessions`.

SQLListener Spark Listener

`SQLListener` is a custom [SparkListener](#) that collects information about SQL query executions for web UI (to display in [SQL tab](#)). It relies on `spark.sql.execution.id` key to distinguish between queries.

Internally, it uses [SQLExecutionUIData](#) data structure exclusively to record all the necessary data for a single SQL query execution. `SQLExecutionUIData` is tracked in the internal registries, i.e. `activeExecutions`, `failedExecutions`, and `completedExecutions` as well as lookup tables, i.e. `_executionIdToData`, `_jobIdToExecutionId`, and `_stageIdToStageMetrics`.

`SQLListener` starts recording a query execution by intercepting a [SparkListenerSQLExecutionStart](#) event (using [onOtherEvent](#) callback).

`SQLListener` stops recording information about a SQL query execution when [SparkListenerSQLExecutionEnd](#) event arrives.

It defines the other callbacks (from [SparkListener](#) interface):

- [onJobStart](#)
- [onJobEnd](#)
- [onExecutorMetricsUpdate](#)
- [onStageSubmitted](#)
- [onTaskEnd](#)

Registering Job and Stages under Active Execution

— `onJobStart` Callback

```
onJobStart(jobStart: SparkListenerJobStart): Unit
```

`onJobStart` reads the `spark.sql.execution.id` key, the identifiers of the job and the stages and then updates the [SQLExecutionUIData](#) for the execution id in `activeExecutions` internal registry.

Note	When <code>onJobStart</code> is executed, it is assumed that SQLExecutionUIData has already been created and available in the internal <code>activeExecutions</code> registry.
------	--

The job in `SQLExecutionUIData` is marked as running with the stages added (to `stages`). For each stage, a `SQLStageMetrics` is created in the internal `_stageIdToStageMetrics` registry. At the end, the execution id is recorded for the job id in the internal `_jobIdToExecutionId`.

onOtherEvent Callback

In `onOtherEvent`, `SQLListener` listens to the following `SparkListenerEvent` events:

- `SparkListenerSQLExecutionStart`
- `SparkListenerSQLExecutionEnd`
- `SparkListenerDriverAccumUpdates`

Registering Active Execution

— `SparkListenerSQLExecutionStart` Event

```
case class SparkListenerSQLExecutionStart(
    executionId: Long,
    description: String,
    details: String,
    physicalPlanDescription: String,
    sparkPlanInfo: SparkPlanInfo,
    time: Long)
extends SparkListenerEvent
```

`SparkListenerSQLExecutionStart` events starts recording information about the `executionId` SQL query execution.

When a `SparkListenerSQLExecutionStart` event arrives, a new `SQLExecutionUIData` for the `executionId` query execution is created and stored in `activeExecutions` internal registry. It is also stored in `_executionIdToData` lookup table.

SparkListenerSQLExecutionEnd Event

```
case class SparkListenerSQLExecutionEnd(
    executionId: Long,
    time: Long)
extends SparkListenerEvent
```

`SparkListenerSQLExecutionEnd` event stops recording information about the `executionId` SQL query execution (tracked as `SQLExecutionUIData`). `SQLListener` saves the input time as `completionTime`.

If there are no other running jobs (registered in `SQLExecutionUIData`), the query execution is removed from the `activeExecutions` internal registry and moved to either `completedExecutions` or `failedExecutions` registry.

This is when `SQLListener` checks the number of `SQLExecutionUIData` entires in either registry — `failedExecutions` or `completedExecutions` — and removes the excess of the old entries beyond `spark.sql.ui.retainedExecutions` Spark property.

SparkListenerDriverAccumUpdates Event

```
case class SparkListenerDriverAccumUpdates(
    executionId: Long,
    accumUpdates: Seq[(Long, Long)])
extends SparkListenerEvent
```

When `SparkListenerDriverAccumUpdates` comes, `SQLExecutionUIData` for the input `executionId` is looked up (in `_executionIdToData`) and `SQLExecutionUIData.driverAccumUpdates` is updated with the input `accumUpdates`.

onJobEnd Callback

```
onJobEnd(jobEnd: SparkListenerJobEnd): Unit
```

When called, `onJobEnd` retrieves the `SQLExecutionUIData` for the job and records it either successful or failed depending on the job result.

If it is the last job of the query execution (tracked as `SQLExecutionUIData`), the execution is removed from `activeExecutions` internal registry and moved to either

If the query execution has already been marked as completed (using `completionTime`) and there are no other running jobs (registered in `SQLExecutionUIData`), the query execution is removed from the `activeExecutions` internal registry and moved to either `completedExecutions` or `failedExecutions` registry.

This is when `SQLListener` checks the number of `SQLExecutionUIData` entires in either registry — `failedExecutions` or `completedExecutions` — and removes the excess of the old entries beyond `spark.sql.ui.retainedExecutions` Spark property.

Getting SQL Execution Data — `getExecution` Method

```
getExecution(executionId: Long): Option[SQLExecutionUIData]
```

Getting Execution Metrics — `getExecutionMetrics` Method

```
getExecutionMetrics(executionId: Long): Map[Long, String]
```

`getExecutionMetrics` gets the metrics (aka *accumulator updates*) for `executionId` (by which it collects all the tasks that were used for an execution).

It is exclusively used to render the [ExecutionPage](#) page in web UI.

`mergeAccumulatorUpdates` Method

`mergeAccumulatorUpdates` is a `private` helper method for...TK

It is used exclusively in [getExecutionMetrics](#) method.

SQLExecutionUIData

`SQLExecutionUIData` is the data abstraction of `SQLListener` to describe SQL query executions. It is a container for jobs, stages, and accumulator updates for a single query execution.

QueryExecutionListener

```
QueryExecutionListener is...FIXME
```

SQLAppStatusListener Spark Listener

`SQLAppStatusListener` is a [SparkListener](#) that...FIXME

Table 1. SQLAppStatusListener's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>liveUpdatePeriodNs</code>	
<code>liveExecutions</code>	
<code>stageMetrics</code>	
<code>uiInitialized</code>	

onExecutionStart Internal Method

```
onExecutionStart(event: SparkListenerSQLExecutionStart): Unit
```

`onExecutionStart` ...FIXME

Note	<code>onExecutionStart</code> is used exclusively when <code>SQLAppStatusListener</code> handles a SparkListenerSQLExecutionStart event.
------	--

onJobStart Callback

```
onJobStart(event: SparkListenerJobStart): Unit
```

Note	<code>onJobStart</code> is part of SparkListener Contract to...FIXME
------	--

`onJobStart` ...FIXME

onStageSubmitted Callback

```
onStageSubmitted(event: SparkListenerStageSubmitted): Unit
```

Note	<code>onStageSubmitted</code> is part of SparkListener Contract to...FIXME
------	--

`onStageSubmitted` ...FIXME

onJobEnd Callback

```
onJobEnd(event: SparkListenerJobEnd): Unit
```

Note	onJobEnd is part of SparkListener Contract to...FIXME
------	---

onJobEnd ...FIXME

onExecutorMetricsUpdate Callback

```
onExecutorMetricsUpdate(event: SparkListenerExecutorMetricsUpdate): Unit
```

Note	onExecutorMetricsUpdate is part of SparkListener Contract to...FIXME
------	--

onExecutorMetricsUpdate ...FIXME

onTaskEnd Callback

```
onTaskEnd(event: SparkListenerTaskEnd): Unit
```

Note	onTaskEnd is part of SparkListener Contract to...FIXME
------	--

onTaskEnd ...FIXME

Handling SparkListenerEvent — onOtherEvent Callback

```
onOtherEvent(event: SparkListenerEvent): Unit
```

Note	onOtherEvent is part of SparkListener Contract to...FIXME
------	---

onOtherEvent ...FIXME

SQLAppStatusPlugin

SQLAppStatusPlugin is a AppStatusPlugin ...FIXME

setupUI Method

```
setupUI(ui: SparkUI): Unit
```

Note

setupUI is part of [AppStatusPlugin Contract](#) to...FIXME.

setupUI ...FIXME

SQLAppStatusStore

`SQLAppStatusStore` is...FIXME

`SQLAppStatusStore` is created when [SQLAppStatusListener](#) or [SQLAppStatusPlugin](#) create a [SQLTab](#).

Creating SQLAppStatusStore Instance

`SQLAppStatusStore` takes the following when created:

- `KVStore`
- Optional [SQLAppStatusListener](#)

WriteTaskStats

`WriteTaskStats` is the no-method contract of [data statistics](#) collected during a Write Task.

Note	BasicWriteTaskStats is the one and only known implementation of the WriteTaskStats Contract in Apache Spark.
------	--

[BasicWriteTaskStats](#) is the one and only known implementation of the [WriteTaskStats Contract](#) in Apache Spark.

BasicWriteTaskStats

`BasicWriteTaskStats` is a basic [WriteTaskStats](#) that carries the following statistics:

- `numPartitions`
- `numFiles`
- `numBytes`
- `numRows`

`BasicWriteTaskStats` is [created](#) exclusively when `BasicWriteTaskStatsTracker` is requested for [getFinalStats](#).

WriteTaskStatsTracker

`WriteTaskStatsTracker` is the abstraction of `WriteTaskStatsTrackers` that collect the statistics of the number of `buckets`, `files`, `partitions` and `rows` processed.

Table 1. WriteTaskStatsTracker Contract

Method	Description
<code>getFinalStats</code>	<pre>getFinalStats(): WriteTaskStats</pre> <p>The final <code>WriteTaskStats</code> statistics computed so far Used when <code>EmptyDirectoryWriteTask</code>, <code>SingleDirectoryWriteTask</code> and <code>DynamicPartitionWriteTask</code> are requested to execute</p>
<code>newBucket</code>	<pre>newBucket(bucketId: Int): Unit</pre> <p>Used when...FIXME</p>
<code>newFile</code>	<pre>newFile(filePath: String): Unit</pre> <p>Used when...FIXME</p>
<code>newPartition</code>	<pre>newPartition(partitionValues: InternalRow): Unit</pre> <p>Used when...FIXME</p>
<code>newRow</code>	<pre>newRow(row: InternalRow): Unit</pre> <p>Used when...FIXME</p>
Note	<p><code>BasicWriteTaskStatsTracker</code> is the one and only known implementation of the <code>WriteTaskStatsTracker Contract</code> in Apache Spark.</p>

BasicWriteTaskStatsTracker

`BasicWriteTaskStatsTracker` is a concrete [WriteTaskStatsTracker](#).

`BasicWriteTaskStatsTracker` is [created](#) exclusively when `BasicWriteJobStatsTracker` is requested for [one](#).

`BasicWriteTaskStatsTracker` takes a Hadoop `Configuration` when created.

Getting Final WriteTaskStats — `getFinalStats` Method

```
getFinalStats(): WriteTaskStats
```

Note

`getFinalStats` is part of the [WriteTaskStatsTracker Contract](#) to get the final [WriteTaskStats](#) statistics computed so far.

`getFinalStats` ...FIXME

WriteJobStatsTracker

`WriteJobStatsTracker` is the abstraction of `WriteJobStatsTrackers` that can `create a new WriteTaskStatsTracker` and `processStats`.

Table 1. WriteJobStatsTracker Contract

Method	Description
<code>newTaskInstance</code>	<pre>newTaskInstance(): WriteTaskStatsTracker</pre> <p>Creates a new <code>WriteTaskStatsTracker</code> Used when <code>EmptyDirectoryWriteTask</code>, <code>SingleDirectoryWriteTask</code> and <code>DynamicPartitionWriteTask</code> are requested for the statsTrackers</p>
<code>processStats</code>	<pre>processStats(stats: Seq[WriteTaskStats]): Unit</pre> <p>Used when...FIXME</p>
Note	<code>BasicWriteJobStatsTracker</code> is the one and only known implementation of the <code>WriteJobStatsTracker Contract</code> in Apache Spark.

BasicWriteJobStatsTracker

`BasicWriteJobStatsTracker` is a concrete [WriteJobStatsTracker](#).

`BasicWriteJobStatsTracker` is [created](#) when [DataWritingCommand](#) and Spark Structured Streaming's `FileStreamSink` are requested for one.

When requested for a new [WriteTaskStatsTracker](#), `BasicwriteJobStatsTracker` creates a new [BasicWriteTaskStatsTracker](#).

Creating BasicWriteJobStatsTracker Instance

`BasicWriteJobStatsTracker` takes the following when created:

- Serializable Hadoop `Configuration`
- Metrics (`Map[String, SQLMetric]`)

Logging

Spark uses [log4j](#) for logging.

Logging Levels

The valid logging levels are [log4j's Levels](#) (from most specific to least):

- OFF (most specific, no logging)
- FATAL (most specific, little data)
- ERROR
- WARN
- INFO
- DEBUG
- TRACE (least specific, a lot of data)
- ALL (least specific, all data)

conf/log4j.properties

You can set up the default logging for Spark shell in `conf/log4j.properties`. Use `conf/log4j.properties.template` as a starting point.

Setting Default Log Level Programmatically

Refer to [Setting Default Log Level Programmaticaly](#) in [SparkContext — Entry Point to Spark Core](#).

Setting Log Levels in Spark Applications

In standalone Spark applications or while in [Spark Shell](#) session, use the following:

```
import org.apache.log4j.{Level, Logger}

Logger.getLogger(classOf[RackResolver]).getLevel
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)
```

sbt

When running a Spark application from within sbt using `run` task, you can use the following `build.sbt` to configure logging levels:

```
fork in run := true
javaOptions in run ++= Seq(
  "-Dlog4j.debug=true",
  "-Dlog4j.configuration=log4j.properties")
outputStrategy := Some(StdoutOutput)
```

With the above configuration `log4j.properties` file should be on CLASSPATH which can be in `src/main/resources` directory (that is included in CLASSPATH by default).

When `run` starts, you should see the following output in sbt:

```
[spark-activator]> run
[info] Running StreamingApp
log4j: Trying to find [log4j.properties] using context classloader sun.misc.Launcher$A
ppClassLoader@1b6d3586.
log4j: Using URL [file:/Users/jacek/dev/oss/spark-activator/target/scala-2.11/classes/
log4j.properties] for automatic log4j configuration.
log4j: Reading configuration from URL file:/Users/jacek/dev/oss/spark-activator/target
/scala-2.11/classes/log4j.properties
```

Disabling Logging

Use the following `conf/log4j.properties` to disable logging completely:

```
log4j.logger.org=OFF
```

Spark SQL's Performance Tuning Tips and Tricks (aka Case Studies)

From time to time I'm lucky enough to find ways to optimize structured queries in Spark SQL. These findings (or discoveries) usually fall into a study category than a single topic and so the goal of **Spark SQL's Performance Tuning Tips and Tricks** chapter is to have a single place for the so-called tips and tricks.

1. [Number of Partitions for groupBy Aggregation](#)

Others

1. Avoid `objectType` as it turns whole-stage Java code generation off.
2. Keep [whole-stage codegen requirements](#) in mind, in particular avoid physical operators with `supportCodegen` flag off.

Case Study: Number of Partitions for groupBy Aggregation

Important	<p>As it fairly often happens in my life, right after I had described the discovery I found out I was wrong and the "Aha moment" was gone.</p> <p>Until I thought about the issue again and took the shortest path possible. See Case 4 for the definitive solution.</p> <p>I'm leaving the page with no changes in-between so you can read it and learn from my mistakes.</p>
-----------	--

The goal of the case study is to fine tune the number of partitions used for `groupBy` aggregation.

Given the following 2-partition dataset the task is to write a structured query so there are no empty partitions (or as little as possible).

```
// 2-partition dataset
val ids = spark.range(start = 0, end = 4, step = 1, numPartitions = 2)
scala> ids.show
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
+---+

scala> ids.rdd.toDebugString
res1: String =
(2) MapPartitionsRDD[8] at rdd at <console>:26 []
 | MapPartitionsRDD[7] at rdd at <console>:26 []
 | MapPartitionsRDD[6] at rdd at <console>:26 []
 | MapPartitionsRDD[5] at rdd at <console>:26 []
 | ParallelCollectionRDD[4] at rdd at <console>:26 []
```

Note	<p>By default Spark SQL uses <code>spark.sql.shuffle.partitions</code> number of partitions for aggregations and joins, i.e. <code>200</code> by default.</p> <p>That often leads to explosion of partitions for nothing that does impact the performance of a query since these 200 tasks (per partition) have all to start and finish before you get the result.</p> <p><i>Less is more</i> remember?</p>
------	---

Case 1: Default Number of Partitions — spark.sql.shuffle.partitions Property

This is the moment when you learn that *sometimes* relying on defaults may lead to poor performance.

Think how many partitions the following query really requires?

```
val groupingExpr = 'id % 2 as "group"
val q = ids.
  groupBy(groupingExpr).
  agg(count($"id") as "count")
```

You may have expected to have at most 2 partitions given the number of groups.

Wrong!

```
scala> q.explain
== Physical Plan ==
*HashAggregate(keys=[(id#0L % 2)#17L], functions=[count(1)])
+- Exchange hashpartitioning((id#0L % 2)#17L, 200)
  +- *HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#17L], functions=[partial_count(1)])
    +- *Range (0, 4, step=1, splits=2)

scala> q.rdd.toDebugString
res5: String =
(200) MapPartitionsRDD[16] at rdd at <console>:30 []
|  MapPartitionsRDD[15] at rdd at <console>:30 []
|  MapPartitionsRDD[14] at rdd at <console>:30 []
|  ShuffledRowRDD[13] at rdd at <console>:30 []
+- (2) MapPartitionsRDD[12] at rdd at <console>:30 []
  |  MapPartitionsRDD[11] at rdd at <console>:30 []
  |  MapPartitionsRDD[10] at rdd at <console>:30 []
  |  ParallelCollectionRDD[9] at rdd at <console>:30 []
```

When you execute the query you should see 200 or so partitions in use in web UI.

```
scala> q.show
+-----+
|group|count|
+-----+
|    0|    2|
|    1|    2|
+-----+
```

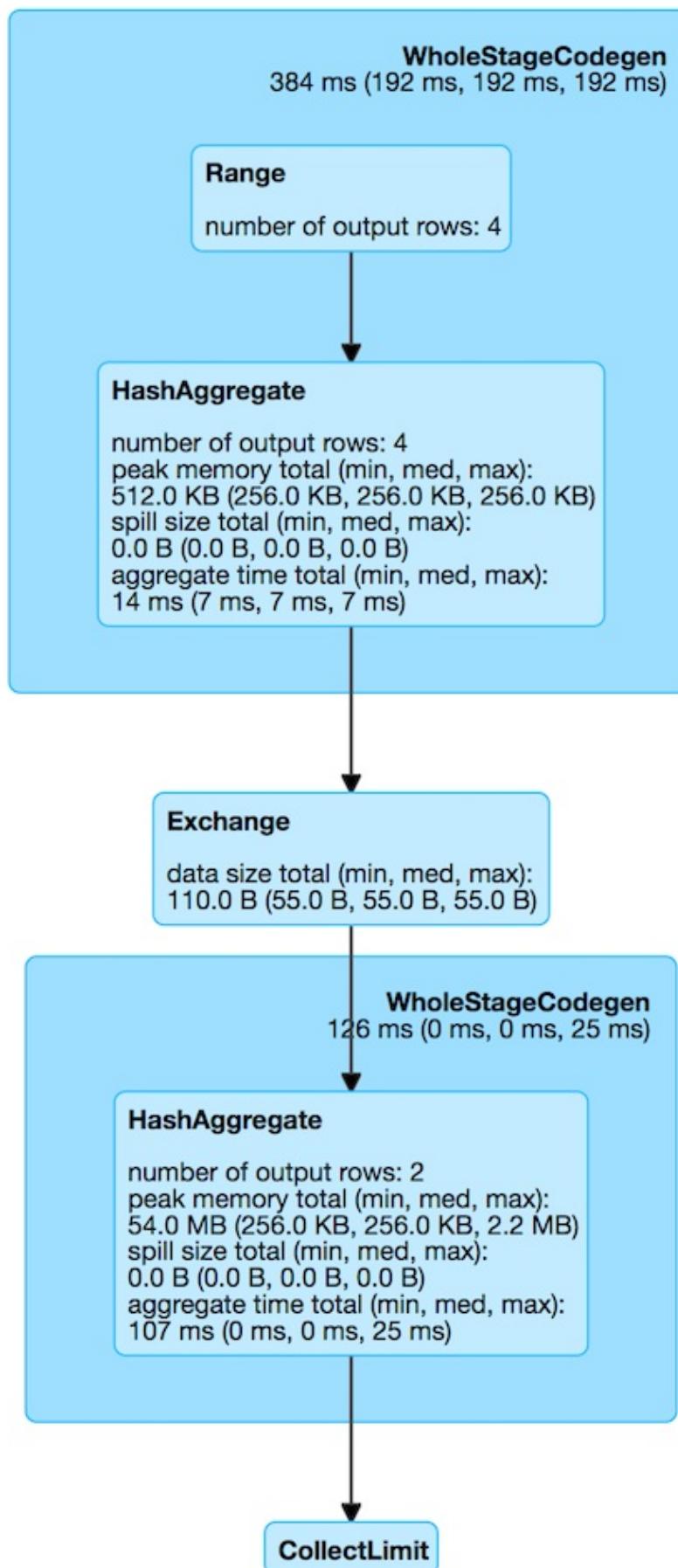
Succeeded Jobs: [2](#) [3](#) [4](#) [5](#) [6](#)

Figure 1. Case 1's Physical Plan with Default Number of Partitions

Note	The number of Succeeded Jobs is 5.
------	---

Case 2: Using repartition Operator

Let's rewrite the query to use `repartition` operator.

`repartition` operator is indeed a step in the right direction when used with caution as it may lead to an unnecessary shuffle (aka exchange in Spark SQL's parlance).

Think how many partitions the following query really requires?

```
val groupingExpr = 'id % 2 as "group"
val q = ids.
    repartition(groupingExpr). // <-- repartition per groupBy expression
    groupBy(groupingExpr).
    agg(count($"id") as "count")
```

You may have expected 2 partitions again?!

Wrong!

```
scala> q.explain
== Physical Plan ==
*HashAggregate(keys=[(id#6L % 2)#105L], functions=[count(1)])
+- Exchange hashpartitioning((id#6L % 2)#105L, 200)
   +- *HashAggregate(keys=[(id#6L % 2) AS (id#6L % 2)#105L], functions=[partial_count(1)])
      +- Exchange hashpartitioning((id#6L % 2), 200)
         +- *Range (0, 4, step=1, splits=2)

scala> q.rdd.toDebugString
res1: String =
(200) MapPartitionsRDD[57] at rdd at <console>:30 []
 |  MapPartitionsRDD[56] at rdd at <console>:30 []
 |  MapPartitionsRDD[55] at rdd at <console>:30 []
 |  ShuffledRowRDD[54] at rdd at <console>:30 []
+- (200) MapPartitionsRDD[53] at rdd at <console>:30 []
   |  MapPartitionsRDD[52] at rdd at <console>:30 []
   |  ShuffledRowRDD[51] at rdd at <console>:30 []
   +- (2) MapPartitionsRDD[50] at rdd at <console>:30 []
     |  MapPartitionsRDD[49] at rdd at <console>:30 []
     |  MapPartitionsRDD[48] at rdd at <console>:30 []
     |  ParallelCollectionRDD[47] at rdd at <console>:30 []
```

Compare the physical plans of the two queries and you will surely regret using `repartition` operator in the latter as you *did* cause an extra shuffle stage (!)

Case 3: Using repartition Operator With Explicit Number of Partitions

The discovery of the day is to notice that `repartition` operator accepts an additional parameter for...the number of partitions (!)

As a matter of fact, there are two variants of `repartition` operator with the number of partitions and the trick is to use the one with partition expressions (that will be used for grouping as well as...hash partitioning).

```
repartition(numPartitions: Int, partitionExprs: Column*): Dataset[T]
```

Can you think of the number of partitions the following query uses? I'm sure you have guessed correctly!

```
val groupingExpr = 'id % 2 as "group"
val q = ids.
    repartition(numPartitions = 2, groupingExpr). // <-- repartition per groupBy expression
    groupBy(groupingExpr).
    agg(count($"id") as "count")
```

You may have expected 2 partitions again?!

Correct!

```

scala> q.explain
== Physical Plan ==
*HashAggregate(keys=[(id#6L % 2)#129L], functions=[count(1)])
+- Exchange hashpartitioning((id#6L % 2)#129L, 200)
   +- *HashAggregate(keys=[(id#6L % 2) AS (id#6L % 2)#129L], functions=[partial_count(1)])
      +- Exchange hashpartitioning((id#6L % 2), 2)
         +- *Range (0, 4, step=1, splits=2)

scala> q.rdd.toDebugString
res14: String =
(200) MapPartitionsRDD[78] at rdd at <console>:30 []
|  MapPartitionsRDD[77] at rdd at <console>:30 []
|  MapPartitionsRDD[76] at rdd at <console>:30 []
|  ShuffledRowRDD[75] at rdd at <console>:30 []
+- (2) MapPartitionsRDD[74] at rdd at <console>:30 []
   |  MapPartitionsRDD[73] at rdd at <console>:30 []
   |  ShuffledRowRDD[72] at rdd at <console>:30 []
+- (2) MapPartitionsRDD[71] at rdd at <console>:30 []
   |  MapPartitionsRDD[70] at rdd at <console>:30 []
   |  MapPartitionsRDD[69] at rdd at <console>:30 []
   |  ParallelCollectionRDD[68] at rdd at <console>:30 []

```



Congratulations! You are done.

Not quite. Read along!

Case 4: Remember `spark.sql.shuffle.partitions` Property? Set It Up Properly

```

import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, 2)
// spark.conf.set(SHUFFLE_PARTITIONS.key, 2)

scala> spark.sessionState.conf.numShufflePartitions
res8: Int = 2

val q = ids.
  groupBy(groupingExpr).
  agg(count($"id") as "count")

```

Number of Partitions for groupBy Aggregation

```
scala> q.explain
== Physical Plan ==
*HashAggregate(keys=[(id#0L % 2)#40L], functions=[count(1)])
+- Exchange hashpartitioning((id#0L % 2)#40L, 2)
   +- *HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#40L], functions=[partial_count(1)])
      +- *Range (0, 4, step=1, splits=2)

scala> q.rdd.toDebugString
res10: String =
(2) MapPartitionsRDD[31] at rdd at <console>:31 []
|  MapPartitionsRDD[30] at rdd at <console>:31 []
|  MapPartitionsRDD[29] at rdd at <console>:31 []
|  ShuffledRowRDD[28] at rdd at <console>:31 []
+- (2) MapPartitionsRDD[27] at rdd at <console>:31 []
   |  MapPartitionsRDD[26] at rdd at <console>:31 []
   |  MapPartitionsRDD[25] at rdd at <console>:31 []
   |  ParallelCollectionRDD[24] at rdd at <console>:31 []
```

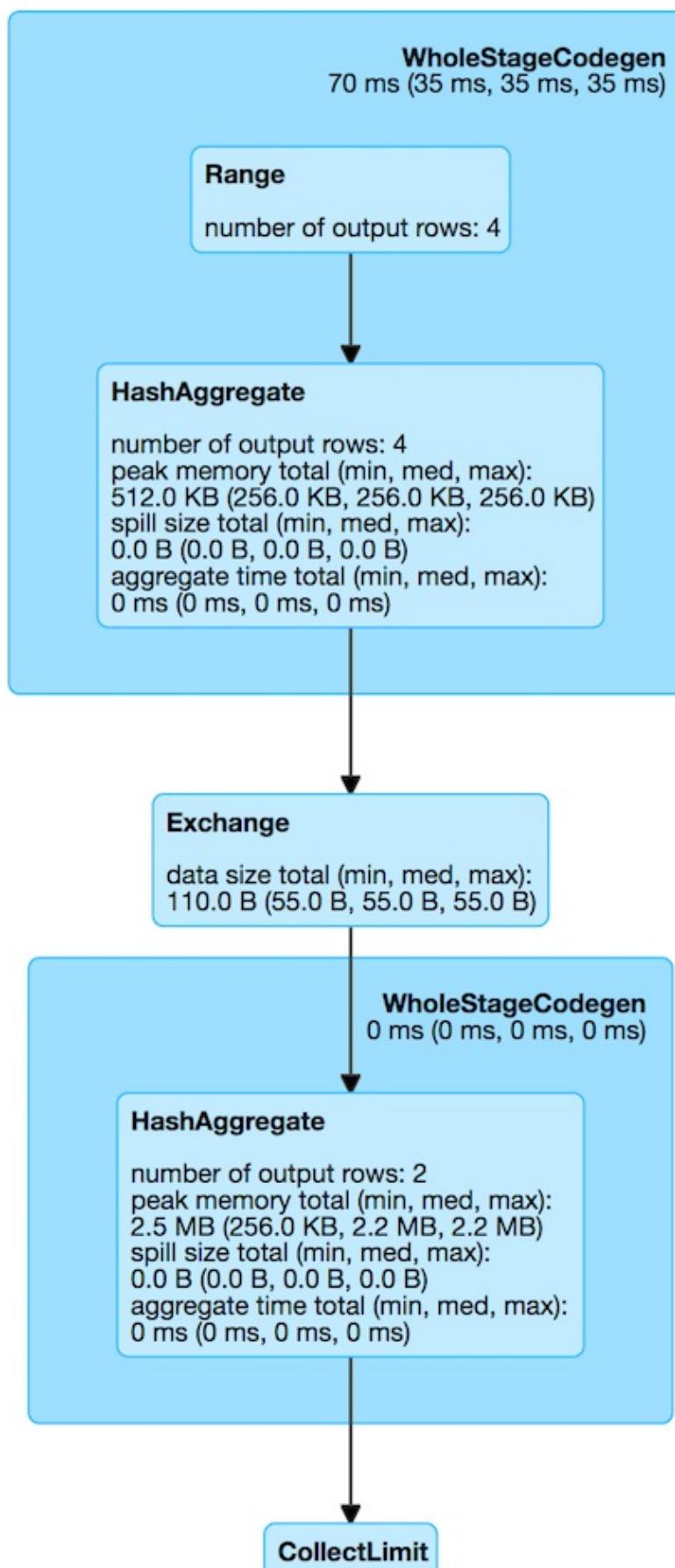
Succeeded Jobs: 7 8

Figure 2. Case 4's Physical Plan with Custom Number of Partitions

Note	The number of Succeeded Jobs is 2.
------	---

Congratulations! You are done now.

Debugging Query Execution

`debug` package object contains [tools](#) for **debugging query execution**, i.e. a full analysis of structured queries (as [Datasets](#)).

Table 1. Debugging Query Execution Tools (debug Methods)

Method	Description
<code>debug</code>	Debugging a structured query <code>debug(): Unit</code>
<code>debugCodegen</code>	Displays the Java source code generated for a structured query in whole-stage code generation (i.e. the output of each WholeStageCodegen subtree in a query plan). <code>debugCodegen(): Unit</code>

`debug` package object is in `org.apache.spark.sql.execution.debug` package that you have to import before you can use the `debug` and `debugCodegen` methods.

```
// Import the package object
import org.apache.spark.sql.execution.debug._

// Every Dataset (incl. DataFrame) has now the debug and debugCodegen methods
val q: DataFrame = ...
q.debug
q.debugCodegen
```

Tip	Read up on Package Objects in the Scala programming language.
-----	---

Internally, `debug` package object uses `DebugQuery` implicit class that "extends" `Dataset[_]` Scala type with the `debug methods`.

```
implicit class DebugQuery(query: Dataset[_]) {
  def debug(): Unit = ...
  def debugCodegen(): Unit = ...
}
```

Tip	Read up on Implicit Classes in the official documentation of the Scala programming language.
-----	--

Debugging Dataset — `debug` Method

```
debug(): Unit
```

`debug` requests the [QueryExecution](#) (of the [structured query](#)) for the [optimized physical query plan](#).

`debug` transforms the optimized physical query plan to add a new [DebugExec](#) physical operator for every physical operator.

`debug` requests the query plan to [execute](#) and then counts the number of rows in the result. It prints out the following message:

```
Results returned: [count]
```

In the end, `debug` requests every [DebugExec](#) physical operator (in the query plan) to [dumpStats](#).

```
val q = spark.range(10).where('id === 4)

scala> :type q
org.apache.spark.sql.Dataset[Long]

// Extend Dataset[Long] with debug and debugCodegen methods
import org.apache.spark.sql.execution.debug._

scala> q.debug
Results returned: 1
== WholeStageCodegen ==
Tuples output: 1
  id LongType: {java.lang.Long}
== Filter (id#0L = 4) ==
Tuples output: 0
  id LongType: {}
== Range (0, 10, step=1, splits=8) ==
Tuples output: 0
  id LongType: {}
```

Displaying Java Source Code Generated for Structured Query in Whole-Stage Code Generation ("Debugging" Codegen) — `debugCodegen` Method

```
debugCodegen(): Unit
```

`debugCodegen` requests the [QueryExecution](#) (of the [structured query](#)) for the [optimized physical query plan](#).

In the end, `debugCodegen` simply [codegenString](#) the query plan and prints it out to the standard output.

```
import org.apache.spark.sql.execution.debug._

scala> spark.range(10).where('id === 4).debugCodegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Filter (id#29L = 4)
+- *Range (0, 10, splits=8)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIterator(references);
/* 003 */
/* 004 */
/* 005 */     final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {
/* 006 */         private Object[] references;
...
}
```

`debugCodegen` is equivalent to using `debug` interface of the [QueryExecution](#).

Note

```
val q = spark.range(1, 1000).select('id+1+2+3, 'id+4+5+6)
scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Project [(id#3L + 6) AS (((id + 1) + 2) + 3)#6L, (id#3L + 15) AS (((id + 4) +
+- *Range (1, 1000, step=1, splits=8)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIterator(references);
/* 003 */
/* 004 */
/* 005 */     final class GeneratedIterator extends org.apache.spark.sql.execution...
```

codegenToSeq Method

`codegenToSeq(): Seq[(String, String)]`

`codegenToSeq ...FIXME`

Note

`codegenToSeq` is used when...FIXME

codegenString Method

```
codegenString(plan: SparkPlan): String
```

codegenString ...FIXME

Note

codegenString is used when...FIXME

Catalyst—Tree Manipulation Framework

Catalyst is an execution-agnostic framework to represent and manipulate a **dataflow graph**, i.e. trees of [relational operators](#) and [expressions](#).

Note

The Catalyst framework were first introduced in [SPARK-1251 Support for optimizing and executing structured queries](#) and became part of Apache Spark on 20/Mar/14 19:12.

The main abstraction in Catalyst is [TreeNode](#) that is then used to build trees of [Expressions](#) or [QueryPlans](#).

Spark 2.0 uses the Catalyst tree manipulation framework to build an extensible **query plan optimizer** with a number of query optimizations.

Catalyst supports both rule-based and cost-based optimization.

TreeNode — Node in Catalyst Tree

`TreeNode` is the [contract](#) of [nodes](#) in [Catalyst](#) tree with [name](#) and zero or more [children](#).

```
package org.apache.spark.sql.catalyst.trees

abstract class TreeNode[BaseType <: TreeNode[BaseType]] extends Product {
  self: BaseType =>

  // only required properties (vals and methods) that have no implementation
  // the others follow
  def children: Seq[BaseType]
  def verboseString: String
}
```

`TreeNode` is a recursive data structure that can have one or many [children](#) that are again `TreeNodes`.

Tip	Read up on <code><:</code> type operator in Scala in Upper Type Bounds .
-----	---

Scala-specific, `TreeNode` is an abstract class that is the [base class](#) of Catalyst Expression and [QueryPlan](#) abstract classes.

`TreeNode` therefore allows for building entire trees of `TreeNodes`, e.g. generic [query plans](#) with concrete [logical](#) and [physical](#) operators that both use [Catalyst expressions](#) (which are `TreeNodes` again).

Note	Spark SQL uses <code>TreeNode</code> for query plans and Catalyst expressions that can further be used together to build more advanced trees, e.g. Catalyst expressions can have query plans as subquery expressions .
------	--

`TreeNode` can itself be a node in a tree or a collection of nodes, i.e. itself and the [children](#) nodes. Not only does `TreeNode` come with the [methods](#) that you may have used in [Scala Collection API](#) (e.g. `map`, `flatMap`, `collect`, `collectFirst`, `foreach`), but also specialized ones for more advanced tree manipulation, e.g. `mapChildren`, `transform`, `transformDown`, `transformUp`, `foreachUp`, `numberedTreeString`, `p`, `asCode`, `prettyJson`.

Table 1. `TreeNode` API (Public Methods)

Method	Description
apply	<code>apply(number: Int): TreeNode[_]</code>

<code>argString</code>	<code>argString: String</code>
<code>asCode</code>	<code>asCode: String</code>
<code>collect</code>	<code>collect[B](pf: PartialFunction[BaseType, B]): Seq[B]</code>
<code>collectFirst</code>	<code>collectFirst[B](pf: PartialFunction[BaseType, B]): Option[B]</code>
<code>collectLeaves</code>	<code>collectLeaves(): Seq[BaseType]</code>
<code>fastEquals</code>	<code>fastEquals(other: TreeNode[_]): Boolean</code>
<code>find</code>	<code>find(f: BaseType => Boolean): Option[BaseType]</code>
<code>flatMap</code>	<code>flatMap[A](f: BaseType => TraversableOnce[A]): Seq[A]</code>
<code>foreach</code>	<code>foreach(f: BaseType => Unit): Unit</code>
<code>foreachUp</code>	<code>foreachUp(f: BaseType => Unit): Unit</code>
<code>generateTreeString</code>	<code>generateTreeString(depth: Int, lastChildren: Seq[Boolean], builder: StringBuilder, verbose: Boolean, prefix: String = "", addSuffix: Boolean = false): StringBuilder</code>
<code>map</code>	<code>map[A](f: BaseType => A): Seq[A]</code>
<code>mapChildren</code>	<code>mapChildren(f: BaseType => BaseType): BaseType</code>

nodeName	nodeName: String
numberedTreeString	numberedTreeString: String
p	p(number: Int): BaseType
prettyJson	prettyJson: String
simpleString	simpleString: String
toJSON	toJSON: String
transform	transform(rule: PartialFunction[BaseType, BaseType]): BaseType
transformDown	transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType
transformUp	transformUp(rule: PartialFunction[BaseType, BaseType]): BaseType
treeString	treeString: String treeString(verbose: Boolean, addSuffix: Boolean = false): String
verboseString	verboseString: String
verboseStringWithSuffix	verboseStringWithSuffix: String
withNewChildren	withNewChildren(newChildren: Seq[BaseType]): BaseType

Table 2. (Subset of) TreeNode Contract

Method	Description
children	Child nodes
verboseString	One-line verbose description Used when <code>TreeNode</code> is requested for <code>generateTreeString</code> (with <code>verbose</code> flag enabled) and <code>verboseStringWithSuffix</code>

Table 3. TreeNodes

TreeNode	Description
Expression	
QueryPlan	

`TreeNode` abstract type is a fairly advanced Scala type definition (at least comparing to the other Scala types in Spark) so understanding its behaviour even outside Spark might be worthwhile by itself.

Tip

```
abstract class TreeNode[BaseType <: TreeNode[BaseType]] extends Product {
  self: BaseType =>
  // ...
}
```

withNewChildren Method

```
withNewChildren(newChildren: Seq[BaseType]): BaseType
```

`withNewChildren ...FIXME`

Note

`withNewChildren` is used when...FIXME

Simple Node Description — `simpleString` Method

```
simpleString: String
```

`simpleString` gives a simple one-line description of a `TreeNode`.

Internally, `simpleString` is the `nodeName` followed by `argString` separated by a single white space.

Note

`simpleString` is used when `TreeNode` is requested for `argString` (of child nodes) and `tree text representation` (with `verbose` flag off).

Numbered Text Representation — `numberedTreeString` Method

```
numberedTreeString: String
```

`numberedTreeString` adds numbers to the `text representation of all the nodes`.

Note

`numberedTreeString` is used primarily for interactive debugging using `apply` and `p` methods.

Getting n-th TreeNode in Tree (for Interactive Debugging) — `apply` Method

```
apply(number: Int): TreeNode[_]
```

`apply` gives `number`-th tree node in a tree.

Note

`apply` can be used for interactive debugging.

Internally, `apply` gets the node at `number` position or `null`.

Getting n-th BaseType in Tree (for Interactive Debugging) — `p` Method

```
p(number: Int): BaseType
```

`p` gives `number`-th tree node in a tree as `BaseType` for interactive debugging.

Note

`p` can be used for interactive debugging.

Note

- `BaseType` is the base type of a tree and in Spark SQL can be:
- `LogicalPlan` for logical plan trees
 - `SparkPlan` for physical plan trees
 - `Expression` for expression trees

Text Representation — `toString` Method

```
toString: String
```

Note

`toString` is part of Java's `Object Contract` for the string representation of an object, e.g. `TreeNode`.

`toString` simply returns the text representation of all nodes in the tree.

Text Representation of All Nodes in Tree — `treeString` Method

```
treeString: String (1)
treeString(verbose: Boolean, addSuffix: Boolean = false): String
```

1. Turns verbose flag on

`treeString` gives the string representation of all the nodes in the `TreeNode`.

```
import org.apache.spark.sql.{functions => f}
val q = spark.range(10).withColumn("rand", f.rand())
val executedPlan = q.queryExecution.executedPlan

val output = executedPlan.treeString(verbose = true)

scala> println(output)
*(1) Project [id#0L, rand(6790207094253656854) AS rand#2]
+- *(1) Range (0, 10, step=1, splits=8)
```

Note

`treeString` is used when:

- `TreeNode` is requested for the numbered text representation and the text representation
- `QueryExecution` is requested for simple, extended and with statistics text representations

Verbose Description with Suffix

— `verboseStringWithSuffix` Method

```
verboseStringWithSuffix: String
```

`verboseStringWithSuffix` simply returns [verbose description](#).

Note

`verboseStringWithSuffix` is used exclusively when `TreeNode` is requested to [generateTreeString](#) (with `verbose` and `addSuffix` flags enabled).

Generating Text Representation of Inner and Regular Child Nodes — `generateTreeString` Method

```
generateTreeString(  
    depth: Int,  
    lastChildren: Seq[Boolean],  
    builder: StringBuilder,  
    verbose: Boolean,  
    prefix: String = "",  
    addSuffix: Boolean = false): StringBuilder
```

Internally, `generateTreeString` appends the following node descriptions per the `verbose` and `addSuffix` flags:

- [verbose description with suffix](#) when both are enabled (i.e. `verbose` and `addSuffix` flags are all `true`)
- [verbose description](#) when `verbose` is enabled (i.e. `verbose` is `true` and `addSuffix` is `false`)
- [simple description](#) when `verbose` is disabled (i.e. `verbose` is `false`)

In the end, `generateTreeString` calls itself recursively for the [innerChildren](#) and the [child nodes](#).

Note

`generateTreeString` is used exclusively when `TreeNode` is requested for [text representation of all nodes in the tree](#).

Inner Child Nodes — `innerChildren` Method

```
innerChildren: Seq[TreeNode[_]]
```

`innerChildren` returns the inner nodes that should be shown as an inner nested tree of this node.

`innerChildren` simply returns an empty collection of `TreeNodes`.

Note

`innerChildren` is used when `TreeNode` is requested to generate the text representation of inner and regular child nodes, `allChildren` and `getNodeNumbered`.

allChildren Property

`allChildren: Set[TreeNode[_]]`

Note

`allChildren` is a Scala lazy value which is computed once when accessed and cached afterwards.

`allChildren ...FIXME`

Note

`allChildren` is used when...FIXME

getNodeNumbered Internal Method

`getNodeNumbered(number: MutableInt): Option[TreeNode[_]]`

`getNodeNumbered ...FIXME`

Note

`getNodeNumbered` is used when...FIXME

foreach Method

`foreach(f: BaseType => Unit): Unit`

`foreach` applies the input function `f` to itself (`this`) first and then (recursively) to the children.

collect Method

`collect[B](pf: PartialFunction[BaseType, B]): Seq[B]`

`collect ...FIXME`

collectFirst Method

```
collectFirst[B](pf: PartialFunction[BaseType, B]): Option[B]
```

collectFirst ...FIXME

collectLeaves Method

```
collectLeaves(): Seq[BaseType]
```

collectLeaves ...FIXME

find Method

```
find(f: BaseType => Boolean): Option[BaseType]
```

find ...FIXME

flatMap Method

```
flatMap[A](f: BaseType => TraversableOnce[A]): Seq[A]
```

flatMap ...FIXME

foreachUp Method

```
foreachUp(f: BaseType => Unit): Unit
```

foreachUp ...FIXME

map Method

```
map[A](f: BaseType => A): Seq[A]
```

map ...FIXME

mapChildren Method

```
mapChildren(f: BaseType => BaseType): BaseType
```

```
mapChildren ...FIXME
```

transform Method

```
transform(rule: PartialFunction[BaseType, BaseType]): BaseType
```

```
transform ...FIXME
```

Transforming Nodes Downwards — transformDown Method

```
transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType
```

```
transformDown ...FIXME
```

transformUp Method

```
transformUp(rule: PartialFunction[BaseType, BaseType]): BaseType
```

```
transformUp ...FIXME
```

asCode Method

```
asCode: String
```

```
asCode ...FIXME
```

prettyJson Method

```
prettyJson: String
```

```
prettyJson ...FIXME
```

Note	<code>prettyJson</code> is used when...FIXME
------	--

toJSON Method

toJSON: String

Note	<code>toJSON</code> is used when...FIXME
------	--

argString Method

argString: String

Note	<code>argString</code> is used when...FIXME
------	---

nodeName Method

nodeName: String

`nodeName` returns the name of the class with `Exec` suffix removed (that is used as a naming convention for the class name of [physical operators](#)).

Note	<code>nodeName</code> is used when <code>TreeNode</code> is requested for simpleString and asCode .
------	---

fastEquals Method

fastEquals(other: TreeNode[_]): Boolean

Note	<code>fastEquals</code> is used when...FIXME
------	--

QueryPlan — Structured Query Plan

`QueryPlan` is part of Catalyst to build a tree of relational operators of a structured query.

Scala-specific, `QueryPlan` is an abstract class that is the base class of `LogicalPlan` and `SparkPlan` (for logical and physical plans, respectively).

A `queryPlan` has an `output` attributes (that serves as the base for the schema), a collection of `expressions` and a `schema`.

`QueryPlan` has `statePrefix` that is used when displaying a plan with `!` to indicate an invalid plan, and `'` to indicate an unresolved plan.

A `QueryPlan` is **invalid** if there are missing input attributes and `children` subnodes are non-empty.

A `QueryPlan` is **unresolved** if the column names have not been verified and column types have not been looked up in the `Catalog`.

A `QueryPlan` has zero, one or more Catalyst expressions.

Note	<code>QueryPlan</code> is a tree of operators that have a tree of expressions.
------	--

`QueryPlan` has `references` property that is the attributes that appear in `expressions` from this operator.

QueryPlan Contract

```
abstract class QueryPlan[T] extends TreeNode[T] {
  def output: Seq[Attribute]
  def validConstraints: Set[Expression]
  // FIXME
}
```

Table 1. QueryPlan Contract

Method	Description
<code>validConstraints</code>	
<code>output</code>	Attribute expressions

Transforming Expressions — `transformExpressions` Method

```
transformExpressions(rule: PartialFunction[Expression, Expression]): this.type
```

`transformExpressions` simply executes `transformExpressionsDown` with the input rule.

Note	<code>transformExpressions</code> is used when...FIXME
------	--

Transforming Expressions — `transformExpressionsDown` Method

```
transformExpressionsDown(rule: PartialFunction[Expression, Expression]): this.type
```

`transformExpressionsDown` applies the rule to each expression in the query operator.

Note	<code>transformExpressionsDown</code> is used when...FIXME
------	--

Applying Transformation Function to Each Expression in Query Operator — `mapExpressions` Method

```
mapExpressions(f: Expression => Expression): this.type
```

`mapExpressions` ...FIXME

Note	<code>mapExpressions</code> is used when...FIXME
------	--

Output Schema Attribute Set — `outputSet` Property

```
outputSet: AttributeSet
```

`outputSet` simply returns an `AttributeSet` for the output schema attributes.

Note	<code>outputSet</code> is used when...FIXME
------	---

producedAttributes Property

Caution	
---------	--

Missing Input Attributes — `missingInput` Property

```
def missingInput: AttributeSet
```

`missingInput` are [attributes](#) that are referenced in expressions but not provided by this node's children (as `inputSet`) and are not produced by this node (as `producedAttributes`).

Output Schema — `schema` Property

You can request the schema of a `QueryPlan` using `schema` that builds [StructType](#) from the [output attributes](#).

```
// the query
val dataset = spark.range(3)

scala> dataset.queryExecution.analyzed.schema
res6: org.apache.spark.sql.types.StructType = StructType(StructField(id, LongType, false
))
```

Output Schema Attributes — `output` Property

```
output: Seq[Attribute]
```

`output` is a collection of [Catalyst attribute expressions](#) that represent the result of a projection in a query that is later used to build the output [schema](#).

Note	<code>output</code> property is also called output schema or result schema .
------	--

```
val q = spark.range(3)

scala> q.queryExecution.analyzed.output
res0: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> q.queryExecution.withCachedData.output
res1: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> q.queryExecution.optimizedPlan.output
res2: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> q.queryExecution.sparkPlan.output
res3: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> q.queryExecution.executedPlan.output
res4: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)
```

You can build a `StructType` from `output` collection of attributes using `toStructType` (that is available through the implicit class `AttributeSeq`).

Tip

```
scala> q.queryExecution.analyzed.output.toStructType
res5: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType
```

Simple (Basic) Description with State Prefix

— `simpleString` Method

```
simpleString: String
```

Note

`simpleString` is part of [TreeNode Contract](#) for the simple text description of a tree node.

`simpleString` adds a [state prefix](#) to the node's [simple text description](#).

State Prefix — `statePrefix` Method

```
statePrefix: String
```

Internally, `statePrefix` gives `!` (exclamation mark) when the node is invalid, i.e. `missingInput` is not empty, and the node is a [parent node](#). Otherwise, `statePrefix` gives an empty string.

Note

`statePrefix` is used exclusively when `QueryPlan` is requested for the [simple text node description](#).

Transforming All Expressions

— `transformAllExpressions` Method

```
transformAllExpressions(rule: PartialFunction[Expression, Expression]): this.type
```

`transformAllExpressions` ...FIXME

Note

`transformAllExpressions` is used when...FIXME

Simple (Basic) Description with State Prefix

— `verboseString` Method

```
verboseString: String
```

Note

`verboseString` is part of [TreeNode Contract](#) to...FIXME.

`verboseString` simply returns the [simple \(basic\) description with state prefix](#).

innerChildren Method

```
innerChildren: Seq[QueryPlan[_]]
```

Note

`innerChildren` is part of [TreeNode Contract](#) to...FIXME.

`innerChildren` simply returns the [subqueries](#).

subqueries Method

```
subqueries: Seq[PlanType]
```

`subqueries` ...FIXME

Note

`subqueries` is used when...FIXME

Canonicalizing Query Plan — doCanonicalize Method

```
doCanonicalize(): PlanType
```

`doCanonicalize` ...FIXME

Note

`doCanonicalize` is used when...FIXME

RuleExecutor Contract — Tree Transformation Rule Executor

`RuleExecutor` is the [base](#) of rule executors that are responsible for [executing](#) a collection of batches ([of rules](#)) to transform a [TreeNode](#).

```
package org.apache.spark.sql.catalyst.rules

abstract class RuleExecutor[TreeType <: TreeNode[_]] {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    protected def batches: Seq[Batch]
}
```

Table 1. RuleExecutor Contract

Property	Description
<code>batches</code>	<code>batches: Seq[Batch]</code> Collection of rule batches , i.e. a sequence of a collection of rules with a name and a strategy that <code>RuleExecutor</code> uses when executed

Note	<code>TreeType</code> is the type of the TreeNode implementation that a <code>RuleExecutor</code> can be executed on, i.e. LogicalPlan , SparkPlan , Expression or a combination thereof.
------	---

Table 2. RuleExecutors (Direct Implementations)

RuleExecutor	Description
Analyzer	Logical query plan analyzer
ExpressionCanonicalizer	
Optimizer	Generic logical query plan optimizer

Applying Rule Batches to TreeNode — execute Method

```
execute(plan: TreeType): TreeType
```

`execute` iterates over [rule batches](#) and applies rules sequentially to the input `plan`.

`execute` tracks the number of iterations and the time of executing each rule (with a plan).

When a rule changes a plan, you should see the following TRACE message in the logs:

```
TRACE HiveSessionStateBuilder$$anon$1:  
== Applying Rule [ruleName] ==  
[currentAndModifiedPlansSideBySide]
```

After the number of iterations has reached the number of iterations for the batch's `strategy` it stops execution and prints out the following WARN message to the logs:

```
WARN HiveSessionStateBuilder$$anon$1: Max iterations ([iteration]) reached for batch [  
batchName]
```

When the plan has not changed (after applying rules), you should see the following TRACE message in the logs and `execute` moves on to applying the rules in the next batch. The moment is called **fixed point** (i.e. when the execution **converges**).

```
TRACE HiveSessionStateBuilder$$anon$1: Fixed point reached for batch [batchName] after  
[iteration] iterations.
```

After the batch finishes, if the plan has been changed by the rules, you should see the following DEBUG message in the logs:

```
DEBUG HiveSessionStateBuilder$$anon$1:  
== Result of Batch [batchName] ==  
[currentAndModifiedPlansSideBySide]
```

Otherwise, when the rules had no changes to a plan, you should see the following TRACE message in the logs:

```
TRACE HiveSessionStateBuilder$$anon$1: Batch [batchName] has no effect.
```

Rule Batch — Collection of Rules

`Batch` is a named collection of `rules` with a `strategy`.

`Batch` takes the following when created:

- Batch name
- `Strategy`

- Collection of [rules](#)

Batch Execution Strategy

`Strategy` is the base of the batch execution strategies that indicate the maximum number of executions (aka *maxIterations*).

```
abstract class Strategy {
    def maxIterations: Int
}
```

Table 3. Strategies

Strategy	Description
<code>Once</code>	A strategy that runs only once (with <code>maxIterations as 1</code>)
<code>FixedPoint</code>	A strategy that runs until fix point (i.e. converge) or <code>maxIterations</code> times, whichever comes first

isPlanIntegral Method

```
isPlanIntegral(plan: TreeType): Boolean
```

`isPlanIntegral` simply returns `true`.

Note

`isPlanIntegral` is used exclusively when `RuleExecutor` is requested to [execute](#).

Catalyst Rule — Named Transformation of TreeNodes

`Rule` is a [named](#) transformation that can be [applied](#) to (i.e. [executed on](#) or [transform](#)) a [TreeNode](#) to produce a new [TreeNode](#).

```
package org.apache.spark.sql.catalyst.rules

abstract class Rule[TreeType <: TreeNode[_]] {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def apply(plan: TreeType): TreeType
}
```

Note	<code>TreeType</code> is the type of the TreeNode implementation that a <code>Rule</code> can be applied to, i.e. LogicalPlan , SparkPlan or Expression or a combination thereof.
------	---

`Rule` has a **rule name** (that is the class name of a rule).

```
ruleName: String
```

`Rule` is mainly used to create a [batch of rules](#) for a [RuleExecutor](#).

The other notable use cases of `Rule` are as follows:

- [SparkSessionExtensions](#)
- When `ExperimentalMethods` is requested for [extraOptimizations](#)
- When `BaseSessionStateBuilder` is requested for [customResolutionRules](#), [customPostHocResolutionRules](#), [customOperatorOptimizationRules](#), and the [Optimizer](#)
- When `Analyzer` is requested for [extendedResolutionRules](#) and [postHocResolutionRules](#) (see [BaseSessionStateBuilder](#) and [HiveSessionStateBuilder](#))
- When `optimizer` is requested for [extendedOperatorOptimizationRules](#)
- When `QueryExecution` is requested for [preparations](#)

QueryPlanner — Converting Logical Plan to Physical Trees

`QueryPlanner` plans a logical plan for execution, i.e. converts a logical plan to one or more physical plans using strategies.

Note

`QueryPlanner` generates at least one physical plan.

`QueryPlanner`'s main method is `plan` that defines the extension points, i.e. `strategies`, `collectPlaceholders` and `prunePlans`.

`QueryPlanner` is part of Catalyst Framework.

QueryPlanner Contract

```
abstract class QueryPlanner[PhysicalPlan <: TreeNode[PhysicalPlan]] {
  def collectPlaceholders(plan: PhysicalPlan): Seq[(PhysicalPlan, LogicalPlan)]
  def prunePlans(plans: Iterator[PhysicalPlan]): Iterator[PhysicalPlan]
  def strategies: Seq[GenericStrategy[PhysicalPlan]]
}
```

Table 1. QueryPlanner Contract

Method	Description
<code>strategies</code>	Collection of <code>GenericStrategy</code> planning strategies. Used exclusively as an extension point in <code>plan</code> .
<code>collectPlaceholders</code>	Collection of "placeholder" physical plans and the corresponding <code>logical plans</code> . Used exclusively as an extension point in <code>plan</code> . Overridden in <code>SparkPlanner</code>
<code>prunePlans</code>	Prunes physical plans (e.g. bad or somehow incorrect plans). Used exclusively as an extension point in <code>plan</code> .

Planning Logical Plan — `plan` Method

```
plan(plan: LogicalPlan): Iterator[PhysicalPlan]
```

`plan` converts the input `plan` [logical plan](#) to zero or more `PhysicalPlan` [plans](#).

Internally, `plan` applies [planning strategies](#) to the input `plan` (one by one collecting all as the plan candidates).

`plan` then walks over the plan candidates to [collect placeholders](#).

If a plan does not contain a placeholder, the plan is returned as is. Otherwise, `plan` walks over placeholders (as pairs of `PhysicalPlan` and unplanned [logical plan](#)) and (recursively) [plans](#) the child logical plan. `plan` then replaces the placeholders with the planned child logical plan.

In the end, `plan` [prunes "bad" physical plans](#).

Note	<code>plan</code> is used exclusively (through the concrete SparkPlanner) when a <code>QueryExecution</code> is requested for a physical plan .
------	--

`plan` is used exclusively (through the concrete [SparkPlanner](#)) when a `QueryExecution` [is requested for a physical plan](#).

GenericStrategy

Executing Planning Strategy — apply Method

Caution	FIXME
---------	-------

Tungsten Execution Backend (Project Tungsten)

The goal of **Project Tungsten** is to improve Spark execution by optimizing Spark jobs for **CPU and memory efficiency** (as opposed to network and disk I/O which are considered fast enough). Tungsten focuses on the hardware architecture of the platform Spark runs on, including but not limited to JVM, LLVM, GPU, NVRAM, etc. It does so by offering the following optimization features:

1. [Off-Heap Memory Management](#) using binary in-memory data representation aka **Tungsten row format** and managing memory explicitly,
2. [Cache Locality](#) which is about cache-aware computations with cache-aware layout for high cache hit rates,
3. [Whole-Stage Code Generation](#) (aka *CodeGen*).

Important	Project Tungsten uses <code>sun.misc.unsafe</code> API for direct memory access to bypass the JVM in order to avoid garbage collection.
-----------	---

```
// million integers
val intsMM = 1 to math.pow(10, 6).toInt

// that gives ca 3.8 MB in memory
scala> sc.parallelize(intsMM).cache.count
res0: Long = 1000000

// that gives ca 998.4 KB in memory
scala> intsMM.toDF.cache.count
res1: Long = 1000000
```

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT Storage tab in the web UI. The Storage tab is active, displaying two RDDs: `ParallelCollectionRDD` and `LocalTableScan [value#1]`. Both RDDs have a size of 3.8 MB in memory. The table has columns for RDD Name, Storage Level, Cached Partitions, Fraction Cached, Size in Memory, and Size on Disk. The 'Size in Memory' column is highlighted with a red box.

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
ParallelCollectionRDD	Memory Deserialized 1x Replicated	8	100%	3.8 MB	0.0 B
LocalTableScan [value#1]	Memory Deserialized 1x Replicated	8	100%	998.4 KB	0.0 B

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
ParallelCollectionRDD	Memory Deserialized 1x Replicated	8	100%	3.8 MB	0.0 B
LocalTableScan [value#1]	Memory Deserialized 1x Replicated	8	100%	998.4 KB	0.0 B

Figure 1. RDD vs DataFrame Size in Memory in web UI—Thank you, Tungsten!

Off-Heap Memory Management

Project Tungsten aims at substantially reducing the usage of JVM objects (and therefore JVM garbage collection) by introducing its own off-heap binary memory management. Instead of working with Java objects, Tungsten uses `sun.misc.Unsafe` to manipulate raw memory.

Tungsten uses the compact storage format called [UnsafeRow](#) for data representation that further reduces memory footprint.

Since [Datasets](#) have known [schema](#), Tungsten properly and in a more compact and efficient way lays out the objects on its own. That brings benefits similar to using extensions written in low-level and hardware-aware languages like C or assembler.

It is possible immediately with the data being already serialized (that further reduces or completely avoids serialization between JVM object representation and Spark's internal one).

Cache Locality

Tungsten uses algorithms and **cache-aware data structures** that exploit the physical machine caches at different levels - L1, L2, L3.

Whole-Stage Java Code Generation

Tungsten does code generation at compile time and generates JVM bytecode to access Tungsten-managed memory structures that gives a very fast access. It uses the [Janino compiler](#)—a super-small, super-fast Java compiler.

Note

The code generation was tracked under [SPARK-8159 Improve expression function coverage \(Spark 1.5\)](#).

Tip

[Read Whole-Stage Code Generation](#).

Further Reading and Watching

1. [Project Tungsten: Bringing Spark Closer to Bare Metal](#)
2. (video) [From DataFrames to Tungsten: A Peek into Spark's Future](#) by Reynold Xin (Databricks)
3. (video) [Deep Dive into Project Tungsten: Bringing Spark Closer to Bare Metal](#) by Josh Rosen (Databricks)

InternalRow — Abstract Binary Row Format

Note

`InternalRow` is also called **Catalyst row** or **Spark SQL row**.

Note

`UnsafeRow` is a concrete `InternalRow`.

```
// The type of your business objects
case class Person(id: Long, name: String)

// The encoder for Person objects
import org.apache.spark.sql.Encoders
val personEncoder = Encoders.product[Person]

// The expression encoder for Person objects
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val personExprEncoder = personEncoder.asInstanceOf[ExpressionEncoder[Person]]

// Convert Person objects to InternalRow
scala> val row = personExprEncoder.toRow(Person(0, "Jacek"))
row: org.apache.spark.sql.catalyst.InternalRow = [0,0,1800000005,6b6563614a]

// How many fields are available in Person's InternalRow?
scala> row.numFields
res0: Int = 2

// Are there any NULLs in this InternalRow?
scala> row.isNullAt(0)
res1: Boolean = false

// You can create your own InternalRow objects
import org.apache.spark.sql.catalyst.InternalRow

scala> val ir = InternalRow(5, "hello", (0, "nice"))
ir: org.apache.spark.sql.catalyst.InternalRow = [5,hello,(0,nice)]
```

There are methods to create `InternalRow` objects using the factory methods in the `InternalRow` object.

```
import org.apache.spark.sql.catalyst.InternalRow

scala> InternalRow.empty
res0: org.apache.spark.sql.catalyst.InternalRow = [empty row]

scala> InternalRow(0, "string", (0, "pair"))
res1: org.apache.spark.sql.catalyst.InternalRow = [0,string,(0,pair)]

scala> InternalRow.fromSeq(Seq(0, "string", (0, "pair")))
res2: org.apache.spark.sql.catalyst.InternalRow = [0,string,(0,pair)]
```

getString Method

Caution	FIXME
---------	-------

UnsafeRow — Mutable Raw-Memory Unsafe Binary Row Format

`UnsafeRow` is a concrete `InternalRow` that represents a mutable internal raw-memory (and hence unsafe) binary row format.

In other words, `UnsafeRow` is an `InternalRow` that is backed by raw memory instead of Java objects.

```
// Use ExpressionEncoder for simplicity
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val stringEncoder = ExpressionEncoder[String]
val row = stringEncoder.toRow("hello world")

import org.apache.spark.sql.catalyst.expressions.UnsafeRow
val unsafeRow = row match { case ur: UnsafeRow => ur }

scala> unsafeRow.getBytes
res0: Array[Byte] = Array(0, 0, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 16, 0, 0, 0, 104, 101,
108, 108, 111, 32, 119, 111, 114, 108, 100, 0, 0, 0, 0, 0)

scala> unsafeRow.getUTF8String(0)
res1: org.apache.spark.unsafe.types.UTF8String = hello world
```

`UnsafeRow` knows its **size in bytes**.

```
scala> println(unsafeRow.getSizeInBytes)
32
```

`UnsafeRow` supports Java's `Externalizable` and Kryo's `KryoSerializable` serialization/deserialization protocols.

The fields of a data row are placed using **field offsets**.

`UnsafeRow` considers a `data type` mutable if it is one of the following:

- `BooleanType`
- `ByteType`
- `DateType`
- `DecimalType` (see `isMutable`)
- `DoubleType`

- [FloatType](#)
- [IntegerType](#)
- [LongType](#)
- [NullType](#)
- [ShortType](#)
- [TimestampType](#)

`UnsafeRow` is composed of three regions:

1. Null Bit Set Bitmap Region (1 bit/field) for tracking null values
2. Fixed-Length 8-Byte Values Region
3. Variable-Length Data Section

That gives the property of rows being always 8-byte word aligned and so their size is always a multiple of 8 bytes.

Equality comparison and hashing of rows can be performed on raw bytes since if two rows are identical so should be their bit-wise representation. No type-specific interpretation is required.

isMutable Static Predicate

```
static boolean isMutable(DataType dt)
```

`isMutable` is enabled (`true`) when the input [DataType](#) is among the [mutable field types](#) or a [DecimalType](#).

Otherwise, `isMutable` is disabled (`false`).

	<code>isMutable</code> is used when: <ul style="list-style-type: none"> • <code>UnsafeFixedWidthAggregationMap</code> is requested to supportsAggregationBufferSchema • <code>SortBasedAggregationIterator</code> is requested for newBuffer
Note	

Kryo's KryoSerializable SerDe Protocol

Tip	Read up on KryoSerializable .
-----	---

Serializing JVM Object—KryoSerializable's `write` Method

```
void write(Kryo kryo, Output out)
```

Deserializing Kryo-Managed Object—KryoSerializable's `read` Method

```
void read(Kryo kryo, Input in)
```

Java's Externalizable SerDe Protocol

Tip	Read up on java.io.Externalizable .
-----	---

Serializing JVM Object—Externalizable's `writeExternal` Method

```
void writeExternal(ObjectOutput out)
throws IOException
```

Deserializing Java-Externalized Object—Externalizable's `readExternal` Method

```
void readExternal(ObjectInput in)
throws IOException, ClassNotFoundException
```

pointTo Method

```
void pointTo(Object baseObject, long baseOffset, int sizeInBytes)
```

`pointTo ...FIXME`

Note	<code>pointTo</code> is used when...FIXME
------	---

AggregationIterator — Generic Iterator of UnsafeRows for Aggregate Physical Operators

`AggregationIterator` is the base for [iterators of `UnsafeRows`](#) that...FIXME

Iterators are data structures that allow to iterate over a sequence of elements. They have a `hasNext` method for checking if there is a next element available, and a `next` method which returns the next element and discards it from the iterator.

Table 1. AggregationIterator's Implementations

Name	Description		
<code>ObjectAggregationIterator</code>	Used exclusively when <code>ObjectHashAggregateExec</code> physical operator is executed .		
<code>SortBasedAggregationIterator</code>	Used exclusively when <code>SortAggregateExec</code> physical operator is executed .		
<code>TungstenAggregationIterator</code>	<p>Used exclusively when <code>HashAggregateExec</code> physical operator is executed.</p> <table border="1"> <tr> <td>Note</td><td><code>HashAggregateExec</code> operator is the preferred aggregate physical operator for Aggregation execution planning strategy (over <code>ObjectHashAggregateExec</code> and <code>SortAggregateExec</code>).</td></tr> </table>	Note	<code>HashAggregateExec</code> operator is the preferred aggregate physical operator for Aggregation execution planning strategy (over <code>ObjectHashAggregateExec</code> and <code>SortAggregateExec</code>).
Note	<code>HashAggregateExec</code> operator is the preferred aggregate physical operator for Aggregation execution planning strategy (over <code>ObjectHashAggregateExec</code> and <code>SortAggregateExec</code>).		

Table 2. AggregationIterator's Internal Registries and Counters

Name	Description
aggregateFunctions	Aggregate functions Used when...FIXME
allImperativeAggregateFunctions	ImperativeAggregate functions Used when...FIXME
allImperativeAggregateFunctionPositions	Positions Used when...FIXME
expressionAggInitialProjection	MutableProjection Used when...FIXME
generateOutput	Function used to generate an unsafe row (i.e. <code>(UnsafeRow, InternalRow) → UnsafeRow</code>) Used when: <ul style="list-style-type: none">• <code>ObjectAggregationIterator</code> is requested for the next unsafe row and <code>outputForEmptyGroupingKeyWithoutInput</code>• <code>SortBasedAggregationIterator</code> is requested for the next unsafe row and <code>outputForEmptyGroupingKeyWithoutInput</code>• <code>TungstenAggregationIterator</code> is requested for the next unsafe row and <code>outputForEmptyGroupingKeyWithoutInput</code>
groupingAttributes	Grouping attributes Used when...FIXME
groupingProjection	UnsafeProjection Used when...FIXME
processRow	<code>(InternalRow, InternalRow) → Unit</code> Used when...FIXME

Creating AggregationIterator Instance

`AggregationIterator` takes the following when created:

- Grouping [named expressions](#)
- Input [attributes](#)
- [Aggregate expressions](#)
- Aggregate [attributes](#)
- Initial input buffer offset
- Result [named expressions](#)
- Function to create a new `MutableProjection` given expressions and attributes

`AggregationIterator` initializes the [internal registries and counters](#).

Note	<code>AggregationIterator</code> is a Scala abstract class and cannot be created directly. It is created indirectly for the concrete AggregationIterators .
------	---

initializeAggregateFunctions Internal Method

```
initializeAggregateFunctions(
  expressions: Seq[AggregateExpression],
  startingInputBufferOffset: Int): Array[AggregateFunction]
```

`initializeAggregateFunctions` ...FIXME

Note	<code>initializeAggregateFunctions</code> is used when...FIXME
------	--

generateProcessRow Internal Method

```
generateProcessRow(
  expressions: Seq[AggregateExpression],
  functions: Seq[AggregateFunction],
  inputAttributes: Seq[Attribute]): (InternalRow, InternalRow) => Unit
```

`generateProcessRow` ...FIXME

Note	<code>generateProcessRow</code> is used when...FIXME
------	--

generateResultProjection Method

```
generateResultProjection(): (UnsafeRow, InternalRow) => UnsafeRow
```

```
generateResultProjection ...FIXME
```

	<code>generateResultProjection</code> is used when:
Note	<ul style="list-style-type: none">• <code>AggregationIterator</code> is created• <code>TungstenAggregationIterator</code> is requested for the generateResultProjection

ObjectAggregationIterator

`ObjectAggregationIterator` is...FIXME

next Method

`next(): UnsafeRow`

Note

`next` is part of Scala's [scala.collection.Iterator](#) interface that returns the next element and discards it from the iterator.

`next` ...FIXME

outputForEmptyGroupingKeyWithoutInput Method

`outputForEmptyGroupingKeyWithoutInput(): UnsafeRow`

`outputForEmptyGroupingKeyWithoutInput` ...FIXME

Note

`outputForEmptyGroupingKeyWithoutInput` is used when...FIXME

SortBasedAggregationIterator

`SortBasedAggregationIterator` is...FIXME

next Method

`next(): UnsafeRow`

Note	<code>next</code> is part of Scala's scala.collection.Iterator interface that returns the next element and discards it from the iterator.
------	---

`next` ...FIXME

outputForEmptyGroupingKeyWithoutInput Method

`outputForEmptyGroupingKeyWithoutInput(): UnsafeRow`

`outputForEmptyGroupingKeyWithoutInput` ...FIXME

Note	<code>outputForEmptyGroupingKeyWithoutInput</code> is used when...FIXME
------	---

newBuffer Internal Method

`newBuffer: InternalRow`

`newBuffer` ...FIXME

Note	<code>newBuffer</code> is used when...FIXME
------	---

TungstenAggregationIterator — Iterator of UnsafeRows for HashAggregateExec Physical Operator

`TungstenAggregationIterator` is a `AggregationIterator` that the `HashAggregateExec` aggregate physical operator uses when `executed` (to process `UnsafeRows` per partition and calculate aggregations).

`TungstenAggregationIterator` prefers hash-based aggregation (before switching to sort-based aggregation).

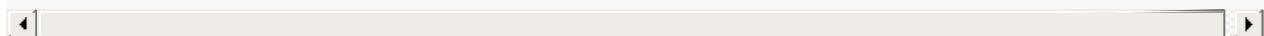
```
val q = spark.range(10).
  groupBy('id % 2 as "group").
  agg(sum("id") as "sum")
val execPlan = q.queryExecution.sparkPlan
scala> println(execPlan.numberedTreeString)
00 HashAggregate(keys=[(id#0L % 2)#11L], functions=[sum(id#0L)], output=[group#3L, sum#7L])
01 +- HashAggregate(keys=[(id#0L % 2) AS (id#0L % 2)#11L], functions=[partial_sum(id#0L)], output=[(id#0L % 2)#11L, sum#13L])
02   +- Range (0, 10, step=1, splits=8)

import org.apache.spark.sql.execution.aggregate.HashAggregateExec
val hashAggExec = execPlan.asInstanceOf[HashAggregateExec]
val hashAggExecRDD = hashAggExec.execute

// MapPartitionsRDD is in private[spark] scope
// Use :paste -raw for the following helper object
package org.apache.spark
object AccessPrivateSpark {
  import org.apache.spark.rdd.RDD
  def mapPartitionsRDD[T](hashAggExecRDD: RDD[T]) = {
    import org.apache.spark.rdd.MapPartitionsRDD
    hashAggExecRDD.asInstanceOf[MapPartitionsRDD[_, _]]
  }
}
// END :paste -raw

import org.apache.spark.AccessPrivateSpark
val mpRDD = AccessPrivateSpark.mapPartitionsRDD(hashAggExecRDD)
val f = mpRDD.iterator(_, _)

import org.apache.spark.sql.execution.aggregate.TungstenAggregationIterator
// FIXME How to show that TungstenAggregationIterator is used?
```



When [created](#), `TungstenAggregationIterator` gets SQL metrics from the `HashAggregateExec` aggregate physical operator being executed, i.e. [numOutputRows](#), [peakMemory](#), [spillSize](#) and [avgHashProbe](#) metrics.

- [numOutputRows](#) is used when `TungstenAggregationIterator` is requested for the [next UnsafeRow](#) (and it [has one](#))
- [peakMemory](#), [spillSize](#) and [avgHashProbe](#) are used at the [end of every task](#) (one per partition)

The metrics are then displayed as part of `HashAggregateExec` aggregate physical operator (e.g. in web UI in [Details for Query](#)).

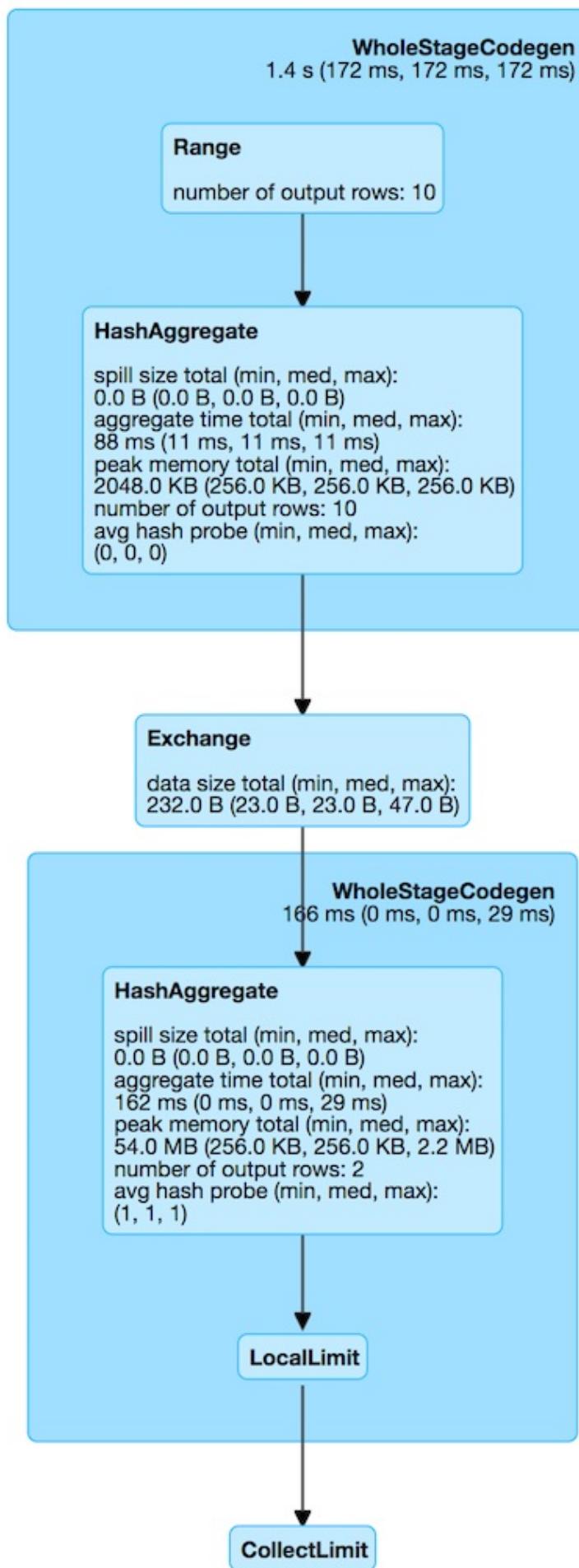


Figure 1. HashAggregateExec in web UI (Details for Query)

Table 1. TungstenAggregationIterator's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
aggregationBufferMapIterator	KVIterator[UnsafeRow, UnsafeRow] Used when...FIXME
hashMap	UnsafeFixedWidthAggregationMap with the following: <ul style="list-style-type: none"> initialAggregationBuffer StructType built from (the aggBufferAttributes of) the aggregate function expressions StructType built from (the attributes of) the groupingExpressions 1024 * 16 initial capacity The page size of the TaskMemoryManager (defaults to spark.buffer.pagesize configuration) Used when TungstenAggregationIterator is requested for the next UnsafeRow, to outputForEmptyGroupingKeyWithoutInput, processInputs, to initialize the aggregationBufferMapIterator and every time a partition has been processed.
initialAggregationBuffer	UnsafeRow that is the aggregation buffer containing initial buffer values. Used when...FIXME
externalSorter	UnsafeKVExternalSorter used for sort-based aggregation
sortBased	Flag to indicate whether TungstenAggregationIterator uses sort-based aggregation (not hash-based aggregation). sortBased flag is disabled (false) by default. Enabled (true) when TungstenAggregationIterator is requested to switch to sort-based aggregation. Used when...FIXME

processInputs Internal Method

```
processInputs(fallbackStartsAt: (Int, Int)): Unit
```

```
processInputs ...FIXME
```

Note

`processInputs` is used exclusively when `TungstenAggregationIterator` is created (and sets the internal flags to indicate whether to use a hash-based aggregation or, in the worst case, a sort-based aggregation when there is not enough memory for groups and their buffers).

Switching to Sort-Based Aggregation (From Preferred Hash-Based Aggregation)

— `switchToSortBasedAggregation` Internal Method

```
switchToSortBasedAggregation(): Unit
```

```
switchToSortBasedAggregation ...FIXME
```

Note

`switchToSortBasedAggregation` is used exclusively when `TungstenAggregationIterator` is requested to `processInputs` (and the `externalSorter` is used).

Getting Next UnsafeRow — `next` Method

```
next(): UnsafeRow
```

Note

`next` is part of Scala's `scala.collection.Iterator` interface that returns the next element and discards it from the iterator.

```
next ...FIXME
```

hasNext Method

```
hasNext: Boolean
```

Note

`hasNext` is part of Scala's `scala.collection.Iterator` interface that tests whether this iterator can provide another element.

```
hasNext ...FIXME
```

Creating TungstenAggregationIterator Instance

`TungstenAggregationIterator` takes the following when created:

- Partition index
- Grouping [named expressions](#)
- Aggregate [expressions](#)
- Aggregate [attributes](#)
- Initial input buffer offset
- Output [named expressions](#)
- Function to create a new `MutableProjection` given Catalyst expressions and attributes
(i.e. `(Seq[Expression], Seq[Attribute]) ⇒ MutableProjection`)
- Output attributes (of the `child` of the `HashAggregateExec` physical operator)
- Iterator of `InternalRows` (from a single partition of the `child` of the `HashAggregateExec` physical operator)
- (used for testing) Optional `HashAggregateExec`'s `testFallbackStartsAt`
- `numOutputRows` [SQLMetric](#)
- `peakMemory` [SQLMetric](#)
- `spillSize` [SQLMetric](#)
- `avgHashProbe` [SQLMetric](#)

Note	The SQL metrics (<code>numOutputRows</code> , <code>peakMemory</code> , <code>spillSize</code> and <code>avgHashProbe</code>) belong to the <code>HashAggregateExec</code> physical operator that created the <code>TungstenAggregationIterator</code> .
------	--

`TungstenAggregationIterator` initializes the [internal registries and counters](#).

`TungstenAggregationIterator` starts processing [input rows](#) and pre-loads the first key-value pair from the `UnsafeFixedWidthAggregationMap` if did not [switch to sort-based aggregation](#).

generateResultProjection Method

```
generateResultProjection(): (UnsafeRow, InternalRow) => UnsafeRow
```

Note	<code>generateResultProjection</code> is part of the AggregationIterator Contract to... FIXME.
------	---

```
generateResultProjection ...FIXME
```

Creating UnsafeRow

— `outputForEmptyGroupingKeyWithoutInput` Method

```
outputForEmptyGroupingKeyWithoutInput(): UnsafeRow
```

```
outputForEmptyGroupingKeyWithoutInput ...FIXME
```

Note

`outputForEmptyGroupingKeyWithoutInput` is used when...FIXME

TaskCompletionListener

`TungstenAggregationIterator` registers a `TaskCompletionListener` that is executed on task completion (for every task that processes a partition).

When executed (once per partition), the `TaskCompletionListener` updates the following metrics:

- `peakMemory`
- `spillSize`
- `avgHashProbe`

CatalystSerde Helper Object

`CatalystSerde` is a Scala object that consists of three utility methods:

1. `deserialize` to create a new logical plan with the input logical plan wrapped inside `DeserializeToObject` logical operator.
2. `serialize`
3. `generateObjAttr`

`CatalystSerde` and belongs to `org.apache.spark.sql.catalyst.plans.logical` package.

Creating Logical Plan with DeserializeToObject Logical Operator for Logical Plan — `deserialize` Method

```
deserialize[T : Encoder](child: LogicalPlan): DeserializeToObject
```

`deserialize` creates a `DeserializeToObject` logical operator for the input `child` logical plan.

Internally, `deserialize` creates a `UnresolvedDeserializer` for the deserializer for the type `T` first and passes it on to a `DeserializeToObject` with a `AttributeReference` (being the result of `generateObjAttr`).

serialize Method

```
serialize[T : Encoder](child: LogicalPlan): SerializeFromObject
```

generateObjAttr Method

```
generateObjAttr[T : Encoder]: Attribute
```

ExternalAppendOnlyUnsafeRowArray — Append-Only Array for UnsafeRows (with Disk Spill Threshold)

`ExternalAppendOnlyUnsafeRowArray` is an append-only array for `UnsafeRows` that spills content to disk when a [predefined spill threshold of rows](#) is reached.

Note Choosing a proper **spill threshold of rows** is a performance optimization.

`ExternalAppendOnlyUnsafeRowArray` is created when:

- `WindowExec` physical operator is [executed](#) (and creates an internal buffer for window frames)
- `WindowFunctionFrame` is [prepared](#)
- `SortMergeJoinExec` physical operator is [executed](#) (and creates a `RowIterator` for INNER and CROSS joins) and for `getBufferedMatches`
- `SortMergeJoinScanner` creates an internal `bufferedMatches`
- `UnsafeCartesianRDD` is computed

Table 1. `ExternalAppendOnlyUnsafeRowArray`'s Internal Registries and Counters

Name	Description
<code>initialSizeOfInMemoryBuffer</code>	FIXME Used when...FIXME
<code>inMemoryBuffer</code>	FIXME Can grow up to <code>numRowsSpillThreshold</code> rows (i.e. new <code>UnsafeRows</code> are added) Used when...FIXME
<code>spillableArray</code>	<code>UnsafeExternalSorter</code> Used when...FIXME
<code>numRows</code>	Used when...FIXME
<code>modificationsCount</code>	Used when...FIXME
<code>numFieldsPerRow</code>	Used when...FIXME

Enable `INFO` logging level for `org.apache.spark.sql.execution.ExternalAppendOnlyUnsafeRowArray` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

Tip

```
log4j.logger.org.apache.spark.sql.execution.ExternalAppendOnlyUnsafeRowArray=INFO
```

Refer to [Logging](#).

generateIterator Method

```
generateIterator(): Iterator[UnsafeRow]  
generateIterator(startIndex: Int): Iterator[UnsafeRow]
```

Caution

FIXME

add Method

```
add(unsafeRow: UnsafeRow): Unit
```

Caution

FIXME

Note

`add` is used when:

- `WindowExec` is executed (and fetches all rows in a partition for a group).
- `SortMergeJoinScanner` buffers matching rows
- `UnsafeCartesianRDD` is computed

clear Method

```
clear(): Unit
```

Caution

FIXME

Creating ExternalAppendOnlyUnsafeRowArray Instance

`ExternalAppendOnlyUnsafeRowArray` takes the following when created:

- [TaskMemoryManager](#)
- [BlockManager](#)
- [SerializerManager](#)
- [TaskContext](#)
- Initial size
- Page size (in bytes)
- Number of rows to hold before spilling them to disk

`ExternalAppendOnlyUnsafeRowArray` initializes the [internal registries and counters](#).

UnsafeFixedWidthAggregationMap

`UnsafeFixedWidthAggregationMap` is a tiny layer (*extension*) around Spark Core's `BytesToBytesMap` to allow for `UnsafeRow` keys and values.

Whenever requested for performance metrics (i.e. [average number of probes per key lookup](#) and [peak memory used](#)), `UnsafeFixedWidthAggregationMap` simply requests the underlying `BytesToBytesMap`.

`UnsafeFixedWidthAggregationMap` is [created](#) when:

- `HashAggregateExec` physical operator is requested to [create a new `UnsafeFixedWidthAggregationMap`](#) (when `HashAggregateExec` physical operator is requested to [generate the Java source code for "produce" path in Whole-Stage Code Generation](#))
- `TungstenAggregationIterator` is [created](#) (when `HashAggregateExec` physical operator is requested to [execute](#) in traditional / non-Whole-Stage-Code-Generation execution path)

Table 1. `UnsafeFixedWidthAggregationMap`'s Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>currentAggregationBuffer</code>	Re-used pointer (as an <code>UnsafeRow</code> with the number of fields to match the <code>aggregationBufferSchema</code>) to the current aggregation buffer Used exclusively when <code>UnsafeFixedWidthAggregationMap</code> is requested to getAggregationBufferFromUnsafeRow .
<code>emptyAggregationBuffer</code>	Empty aggregation buffer (encoded in UnsafeRow format)
<code>groupingKeyProjection</code>	<code>UnsafeProjection</code> for the <code>groupingKeySchema</code> (to encode grouping keys as <code>UnsafeRows</code>)
<code>map</code>	Spark Core's <code>BytesToBytesMap</code> with the <code>taskMemoryManager</code> , <code>initialCapacity</code> , <code>pageSizeBytes</code> and performance metrics enabled

supportsAggregationBufferSchema Static Method

```
boolean supportsAggregationBufferSchema(StructType schema)
```

`supportsAggregationBufferSchema` is a predicate that is enabled (`true`) unless there is a field (in the `fields` of the input schema) whose `data type` is not `mutable`.

Note

The `mutable` data types: `BooleanType`, `ByteType`, `DateType`, `DecimalType`, `DoubleType`, `FloatType`, `IntegerType`, `LongType`, `NullType`, `ShortType` and `TimestampType`.

Examples (possibly all) of data types that are not `mutable`: `ArrayType`, `BinaryType`, `StringType`, `CalendarIntervalType`, `MapType`, `ObjectType` and `StructType`.

```
import org.apache.spark.sql.execution.UnsafeFixedWidthAggregationMap

import org.apache.spark.sql.types._

val schemaWithImmutableField = StructType(StructField("string", StringType) :: Nil)
assert(UnsafeFixedWidthAggregationMap.supportsAggregationBufferSchema(schemaWithImmutableField) == false)

val schemaWithMutableFields = StructType(
  StructField("int", IntegerType) :: StructField("bool", BooleanType) :: Nil)
assert(UnsafeFixedWidthAggregationMap.supportsAggregationBufferSchema(schemaWithMutableFields))
```

Note

`supportsAggregationBufferSchema` is used exclusively when `HashAggregateExec` is requested to `supportsAggregate`.

Creating UnsafeFixedWidthAggregationMap Instance

`UnsafeFixedWidthAggregationMap` takes the following when created:

- Empty aggregation buffer (as an `InternalRow`)
- Aggregation buffer `schema`
- Grouping key `schema`
- Spark Core's `TaskMemoryManager`
- Initial capacity
- Page size (in bytes)

`UnsafeFixedWidthAggregationMap` initializes the `internal registries and counters`.

getAggregationBufferFromUnsafeRow Method

```
UnsafeRow getAggregationBufferFromUnsafeRow(UnsafeRow key) (1)  
UnsafeRow getAggregationBufferFromUnsafeRow(UnsafeRow key, int hash)
```

1. Uses the hash code of the key

`getAggregationBufferFromUnsafeRow` requests the [BytesToBytesMap](#) to `lookup` the input key (to get a `BytesToBytesMap.Location`).

`getAggregationBufferFromUnsafeRow ...FIXME`

Note

- `getAggregationBufferFromUnsafeRow` is used when:
 - `TungstenAggregationIterator` is requested to `processInputs` (exclusively when `TungstenAggregationIterator` is created)
 - (for testing only) `UnsafeFixedWidthAggregationMap` is requested to `getAggregationBuffer`

getAggregationBuffer Method

```
UnsafeRow getAggregationBuffer(InternalRow groupingKey)
```

`getAggregationBuffer ...FIXME`

Note

`getAggregationBuffer` seems to be used exclusively for testing.

Getting KVIterator— iterator Method

```
KVIterator<UnsafeRow, UnsafeRow> iterator()
```

`iterator ...FIXME`

Note

- `iterator` is used when:
 - `HashAggregateExec` physical operator is requested to `finishAggregate`
 - `TungstenAggregationIterator` is created (and pre-loads the first key-value pair from the map)

getPeakMemoryUsedBytes Method

```
long getPeakMemoryUsedBytes()
```

`getPeakMemoryUsedBytes ...FIXME`

Note

- `getPeakMemoryUsedBytes` is used when:
- `HashAggregateExec` physical operator is requested to [finishAggregate](#)
 - `TungstenAggregationIterator` is used in [TaskCompletionListener](#)

getAverageProbesPerLookup Method

`double getAverageProbesPerLookup()`

`getAverageProbesPerLookup ...FIXME`

Note

- `getAverageProbesPerLookup` is used when:
- `HashAggregateExec` physical operator is requested to [finishAggregate](#)
 - `TungstenAggregationIterator` is used in [TaskCompletionListener](#)

free Method

`void free()`

`free ...FIXME`

Note

- `free` is used when:
- `HashAggregateExec` physical operator is requested to [finishAggregate](#)
 - `TungstenAggregationIterator` is requested to [processInputs](#) (when `TungstenAggregationIterator` is created), [get the next UnsafeRow](#), [outputForEmptyGroupingKeyWithoutInput](#) and is created

destructAndCreateExternalSorter Method

`UnsafeKVExternalSorter destructAndCreateExternalSorter() throws IOException`

`destructAndCreateExternalSorter ...FIXME`

	<p><code>destructAndCreateExternalSorter</code> is used when:</p> <ul style="list-style-type: none">• <code>HashAggregateExec</code> physical operator is requested to finishAggregate• <code>TungstenAggregationIterator</code> is requested to processInputs (when <code>TungstenAggregationIterator</code> is created)
--	--

SQL Parsing Framework

SQL Parser Framework in Spark SQL uses ANTLR to translate a SQL text to a [data type](#), [Expression](#), [TableIdentifier](#) or [LogicalPlan](#).

The contract of the SQL Parser Framework is described by [ParserInterface](#) contract. The contract is then abstracted in [AbstractSqlParser](#) class so subclasses have to provide custom [AstBuilder](#) only.

There are two concrete implementations of [AbstractSqlParser](#) :

1. [SparkSqlParser](#) that is the default parser of the SQL expressions into Spark's types.
2. [CatalystSqlParser](#) that is used to parse data types from their canonical string representation.

AbstractSqlParser — Base SQL Parsing Infrastructure

`AbstractSqlParser` is the [base](#) of [ParserInterfaces](#) that use an [AstBuilder](#) to parse SQL statements and convert them to Spark SQL entities, i.e. [DataType](#), [StructType](#), [Expression](#), [LogicalPlan](#) and [TableIdentifier](#).

`AbstractSqlParser` is the foundation of the SQL parsing infrastructure.

```
package org.apache.spark.sql.catalyst.parser

abstract class AbstractSqlParser extends ParserInterface {
    // only required properties (vals and methods) that have no implementation
    // the others follow
    def astBuilder: AstBuilder
}
```

Table 1. `AbstractSqlParser` Contract

Method	Description
<code>astBuilder</code>	AstBuilder for parsing SQL statements. Used in all the <code>parse</code> methods, i.e. parseDataType , parseExpression , parsePlan , parseTableIdentifier , and parseTableSchema .

Table 2. `AbstractSqlParser`'s Implementations

Name	Description
SparkSqlParser	The default SQL parser in SessionState available as <code>sqlParser</code> property. <code>val spark: SparkSession = ...</code> <code>spark.sessionState.sqlParser</code>
CatalystSqlParser	Creates a DataType or a StructType (schema) from their canonical string representation.

Setting Up `SqlBaseLexer` and `SqlBaseParser` for Parsing — `parse` Method

```
parse[T](command: String)(toResult: SqlBaseParser => T): T
```

`parse` sets up a proper ANTLR parsing infrastructure with `SqlBaseLexer` and `SqlBaseParser` (which are the ANTLR-specific classes of Spark SQL that are auto-generated at build time from the `SqlBase.g4` grammar).

Tip

Review the definition of ANTLR grammar for Spark SQL in [sql/catalyst/src/main/antlr4/org/apache/spark/sql/catalyst/parser/SqlBase.g4](https://github.com/apache/spark/blob/v3.1.2/sql/catalyst/src/main/antlr4/org/apache/spark/sql/catalyst/parser/SqlBase.g4).

Internally, `parse` first prints out the following INFO message to the logs:

```
INFO SparkSqlParser: Parsing command: [command]
```

Tip

Enable `INFO` logging level for the custom `AbstractSqlParser`, i.e. `SparkSqlParser` or `CatalystSqlParser`, to see the above INFO message.

`parse` then creates and sets up a `SqlBaseLexer` and `SqlBaseParser` that in turn passes the latter on to the input `toResult` function where the parsing finally happens.

Note

`parse` uses `SLL` prediction mode for parsing first before falling back to `LL` mode.

In case of parsing errors, `parse` reports a `ParseException`.

Note

`parse` is used in all the `parse` methods, i.e. `parseDataType`, `parseExpression`, `parsePlan`, `parseTableIdentifier`, and `parseTableSchema`.

parseDataType Method

```
parseDataType(sqlText: String): DataType
```

Note

`parseDataType` is part of [ParserInterface Contract](#) to...FIXME.

`parseDataType` ...FIXME

parseExpression Method

```
parseExpression(sqlText: String): Expression
```

Note

`parseExpression` is part of [ParserInterface Contract](#) to...FIXME.

`parseExpression` ...FIXME

parseFunctionIdentifier Method

```
parseFunctionIdentifier(sqlText: String): FunctionIdentifier
```

Note `parseFunctionIdentifier` is part of [ParserInterface Contract](#) to...FIXME.

`parseFunctionIdentifier` ...FIXME

parseTableIdentifier Method

```
parseTableIdentifier(sqlText: String): TableIdentifier
```

Note `parseTableIdentifier` is part of [ParserInterface Contract](#) to...FIXME.

`parseTableIdentifier` ...FIXME

parseTableSchema Method

```
parseTableSchema(sqlText: String): StructType
```

Note `parseTableSchema` is part of [ParserInterface Contract](#) to...FIXME.

`parseTableSchema` ...FIXME

parsePlan Method

```
parsePlan(sqlText: String): LogicalPlan
```

Note `parsePlan` is part of [ParserInterface Contract](#) to...FIXME.

`parsePlan` creates a [LogicalPlan](#) for a given SQL textual statement.

Internally, `parsePlan` builds a [SqlBaseParser](#) and requests [AstBuilder](#) to parse a single [SQL statement](#).

If a SQL statement could not be parsed, `parsePlan` throws a [ParseException](#):

```
Unsupported SQL statement
```


AstBuilder — ANTLR-based SQL Parser

`AstBuilder` converts SQL statements into Spark SQL's relational entities (i.e. [data types](#), [Catalyst expressions](#), [logical plans](#) or `TableIdentifiers`) using [visit callback methods](#).

`AstBuilder` is the [AST builder](#) of `AbstractSqlParser` (i.e. the base SQL parsing infrastructure in Spark SQL).

Spark SQL supports SQL statements as described in [SqlBase.g4](#). Using the file can Spark SQL supports at any given time.

"*Almost*" being that although the grammar accepts a SQL statement it can be reported by `AstBuilder`, e.g.

```
scala> sql("EXPLAIN FORMATTED SELECT * FROM myTable").show
org.apache.spark.sql.catalyst.parser.ParseException:
Operation not allowed: EXPLAIN FORMATTED(line 1, pos 0)

== SQL ==
EXPLAIN FORMATTED SELECT * FROM myTable
^^^

at org.apache.spark.sql.catalyst.parser.ParserUtils$.operationNotAllowed(Parse
at org.apache.spark.sql.execution.SparkSqlAstBuilder$$anonfun$visitExplain$1.a
at org.apache.spark.sql.execution.SparkSqlAstBuilder$$anonfun$visitExplain$1.a
at org.apache.spark.sql.catalyst.parser.ParserUtils$.withOrigin(ParserUtils.sc
at org.apache.spark.sql.execution.SparkSqlAstBuilder.visitExplain(SparkSqlPars
at org.apache.spark.sql.execution.SparkSqlAstBuilder.visitExplain(SparkSqlPars
at org.apache.spark.sql.catalyst.parser.SqlBaseParser$ExplainContext.accept(Sq
at org.antlr.v4.runtime.tree.AbstractParseTreeVisitor.visit(AbstractParseTreeV
at org.apache.spark.sql.catalyst.parser.AstBuilder$$anonfun$visitSingleStateme
at org.apache.spark.sql.catalyst.parser.AstBuilder$$anonfun$visitSingleStateme
at org.apache.spark.sql.catalyst.parser.ParserUtils$.withOrigin(ParserUtils.sc
at org.apache.spark.sql.catalyst.parser.AstBuilder.visitSingleStatement(AstBui
at org.apache.spark.sql.catalyst.parser.AbstractSqlParser$$anonfun$parsePlan$1
at org.apache.spark.sql.catalyst.parser.AbstractSqlParser$$anonfun$parsePlan$1
at org.apache.spark.sql.catalyst.parser.AbstractSqlParser.parse(ParseDriver.sc
at org.apache.spark.sql.execution.SparkSqlParser.parse(SparkSqlParser.scala:46
at org.apache.spark.sql.catalyst.parser.AbstractSqlParser.parsePlan(ParseDrive
at org.apache.spark.sql.catalyst.parser.AbstractSqlParser.parsePlan(ParseDrive
at org.apache.spark.sql.SparkSession.sql(SparkSession.scala:617)
... 48 elided
```

Tip

`AstBuilder` is a ANTLR `AbstractParseTreeVisitor` (as `SqlBaseBaseVisitor`) that is generated from [SqlBase.g4](#) ANTLR grammar for Spark SQL.

Note

`SqlBaseBaseVisitor` is a ANTLR-specific base class that is auto-generated at build time from a ANTLR grammar in `SqlBase.g4`.

`SqlBaseBaseVisitor` is an ANTLR [AbstractParseTreeVisitor](#).

Table 1. AstBuilder's Visit Callbacks

Callback Method	ANTLR rule / labeled alternative
-----------------	----------------------------------

visitAliasedQuery				
visitColumnReference				
visitDereference				
visitExists	#exists labeled alternative	Exists expression		
visitExplain	explain rule	<p>ExplainCommand</p> <p>Note</p> <p>Can be a OneRowRelation or DescribeTableCommand statement.</p> <pre>val q = sql("EXPLAIN SELECT * FROM test") scala> println(q.q) scala> println(q.q) 00 ExplainCommand</pre>		
visitFirst	#first labeled alternative	<p>First aggregate function expression</p> <p>FIRST '(' expression (IGNORE)</p>		
visitFromClause	fromClause	<p>LogicalPlan</p> <p>Supports multiple comma-separated relations with optional LATERAL VIEW.</p> <p>A relation can be one of the following:</p> <ul style="list-style-type: none"> • Table identifier • Inline table using VALUES expression • Table-valued function (current catalog) 		
visitFunctionCall	functionCall labeled alternative	<ul style="list-style-type: none"> • UnresolvedFunction for a function • UnresolvedWindowExpression for a WindowSpecReference • WindowExpression for a function <table border="1"> <tr> <td>Tip</td> <td>See the functions section</td> </tr> </table>	Tip	See the functions section
Tip	See the functions section			
		<p>UnresolvedInlineTable unary operator</p> <p>VALUES expression (',' expression)</p>		

visitInlineTable	inlineTable rule	<p><code>expression</code> can be as follows:</p> <ul style="list-style-type: none"> • CreateNamedStruct expression • Any Catalyst expression for <code>tableAlias</code> can be specified even if it's <code>None</code>.
visitInsertIntoTable	#insertIntoTable labeled alternative	<p>InsertIntoTable (indirectly)</p> <p>A 3-element tuple with a <code>TableIdentifier</code>:</p> <pre>INSERT INTO TABLE? tableIdentifier</pre> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>Note insertIntoTable is the labeled alternative in singleInsert</p> </div>
visitInsertOverwriteTable	#insertOverwriteTable labeled alternative	<p>InsertIntoTable (indirectly)</p> <p>A 3-element tuple with a <code>TableIdentifier</code>:</p> <pre>INSERT OVERWRITE TABLE tableIdentifier</pre> <p>In a way, <code>visitInsertOverwriteTable</code> with the <code>exists</code> flag on or off plus <code>partitions</code> are used with no <code>IF</code></p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>Note insertOverwriteTable is the labeled alternative in singleInsert</p> </div>
visitMultiInsertQuery	multiInsertQueryBody	<p>A logical operator with a Insert expression:</p> <pre>FROM relation (',' relation) INSERT OVERWRITE TABLE ...</pre> <pre>FROM relation (',' relation) INSERT INTO TABLE? ...</pre>
visitNamedExpression	namedExpression	<ul style="list-style-type: none"> • <code>Alias</code> (for a single alias) • <code>MultiAlias</code> (for a parenthesized list of aliases) • a bare Expression
visitNamedQuery		SubqueryAlias
		<code>OneRowRelation</code> or LogicalPlan

visitQuerySpecification	querySpecification	Note	visitQuerySpecification clause. val q = sql("select * from test"). scala> println(q.queryString) 00 'Project [unresolved reference: test] +- OneRowRelation[]
visitPredicated	predicated	Expression	
visitRelation	relation	LogicalPlan for a <code>FROM</code> clause.	
visitRowConstructor			
visitSingleDataType	singleDataType	DataType	
visitSingleExpression	singleExpression	Expression Takes the named expression as input	
visitSingleInsertQuery	#singleInsertQuery labeled alternative	A logical operator with a <code>InsertInto</code> clause. INSERT INTO TABLE? tableIdentifier INSERT OVERWRITE TABLE tableIdentifier INTO tableIdentifier	
visitSortItem	sortItem	SortOrder unevaluatable unary expression sortItem : expression ordering=(ASC DESC)? nullOrdering=(FIRST LAST)	
DESC)? (NULLS nullOrder=(LAST	FIRST))? ORDER BY order+=sortItem (',' sortItem)* SORT BY sort+=sortItem (',' sortItem)* (ORDER	SORT) BY sortItem (',' sortItem)*	
visitSingleStatement	singleStatement	LogicalPlan from a single statement	Note A single statement

visitSingleTableIdentifier	singleTableIdentifier	TableIdentifier
visitStar	#star labeled alternative	UnresolvedStar
visitStruct		
visitSubqueryExpression	#subqueryExpression labeled alternative	ScalarSubquery
visitWindowDef	windowDef labeled alternative	<p>WindowSpecDefinition</p> <pre>'(' CLUSTER BY partition+=expr ('((PARTITION DISTRIBUTE ((ORDER SORT) BY sortItem windowFrame?'))'</pre>

Table 2. AstBuilder's Parsing Handlers

Parsing Handler	LogicalPlan Added		
withAggregation	<ul style="list-style-type: none"> GroupingSets for GROUP BY ... GROUPING SETS (...) Aggregate for GROUP BY ... (WITH CUBE WITH ROLLUP)? 		
withGenerate	Generate with a UnresolvedGenerator and join flag turned on for LATERAL VIEW (in SELECT or FROM clauses).		
withHints	<p>Hint for /*+ hint */ in SELECT queries.</p> <table border="1"> <tr> <td>Tip</td> <td>Note + (plus) between /* and */</td> </tr> </table> <p>hint is of the format name or name (param1, param2, ...).</p> <pre>/*+ BROADCAST (table) */</pre>	Tip	Note + (plus) between /* and */
Tip	Note + (plus) between /* and */		
withInsertInto	<ul style="list-style-type: none"> InsertIntoTable for visitSingleInsertQuery or visitMultiInsertQuery InsertIntoDir for...FIXME 		
	<p>Join for a FROM clause and relation alone.</p> <p>The following join types are supported:</p> <ul style="list-style-type: none"> INNER (default) CROSS LEFT (with optional OUTER) 		

	<ul style="list-style-type: none"> • LEFT SEMI • RIGHT (with optional OUTER) • FULL (with optional OUTER) • ANTI (optionally prefixed with LEFT) <p>The following join criteria are supported:</p> <ul style="list-style-type: none"> • ON booleanExpression • USING '(' identifier (',' identifier)* ')' <p>Joins can be NATURAL (with no join criteria).</p>
withQueryResultClauses	
	<p>Adds a query specification to a logical operator.</p> <p>For transform SELECT (with TRANSFORM , MAP OR REDUCE qualifiers), withQuerySpecification does...FIXME</p> <hr/> <p>For regular SELECT (no TRANSFORM , MAP OR REDUCE qualifiers), withQuerySpecification adds (in that order):</p> <ol style="list-style-type: none"> 1. Generate unary logical operators (if used in the parsed SQL text) 2. Filter unary logical plan (if used in the parsed SQL text) 3. GroupingSets or Aggregate unary logical operators (if used in the parsed SQL text) 4. Project and/or Filter unary logical operators 5. WithWindowDefinition unary logical operator (if used in the parsed SQL text) 6. UnresolvedHint unary logical operator (if used in the parsed SQL text)
withQuerySpecification	
withPredicate	<ul style="list-style-type: none"> • NOT? IN '(' query ')' gives an In predicate expression with a ListQuery subquery expression • NOT? IN '(' expression (',' expression)* ')' gives an In predicate expression
	<p>WithWindowDefinition for window aggregates (given WINDOW definitions).</p> <p>Used for withQueryResultClauses and withQuerySpecification with windows definition.</p>

withWindows

```
WINDOW identifier AS windowSpec
      (',' identifier AS windowSpec)*
```

Tip

Consult `windows`, `namedWindow`, `windowSpec`, `windowFrame`, and `frameBound` (with `windowRef` and `windowDef`) ANTLR parsing rules for Spark SQL in [SqlBase.g4](#).

Note

`AstBuilder` belongs to `org.apache.spark.sql.catalyst.parser` package.

Function Examples

The examples are handled by [visitFunctionCall](#).

```
import spark.sessionState.sqlParser

scala> sqlParser.parseExpression("foo()")
res0: org.apache.spark.sql.catalyst.expressions.Expression = 'foo()

scala> sqlParser.parseExpression("foo() OVER windowSpecRef")
res1: org.apache.spark.sql.catalyst.expressions.Expression = unresolvedwindowexpression('foo(), WindowSpecReference(windowSpecRef))

scala> sqlParser.parseExpression("foo() OVER (CLUSTER BY field)")
res2: org.apache.spark.sql.catalyst.expressions.Expression = 'foo() windowspecdefinition('field, UnspecifiedFrame)
```

aliasPlan Internal Method

```
aliasPlan(alias: ParserRuleContext, plan: LogicalPlan): LogicalPlan
```

```
aliasPlan ...FIXME
```

Note

`aliasPlan` is used when...FIXME

mayApplyAliasPlan Internal Method

```
mayApplyAliasPlan(tableAlias: TableAliasContext, plan: LogicalPlan): LogicalPlan
```

```
mayApplyAliasPlan ...FIXME
```

Note

`mayApplyAliasPlan` is used when...FIXME

CatalystSqlParser — DataTypes and StructTypes Parser

`CatalystSqlParser` is a `AbstractSqlParser` with `AstBuilder` as the required `astBuilder`.

`CatalystSqlParser` is used to translate `DataTypes` from their canonical string representation (e.g. when [adding fields to a schema](#) or [casting column to a different data type](#)) or `StructTypes`.

```
import org.apache.spark.sql.types.StructType
scala> val struct = new StructType().add("a", "int")
struct: org.apache.spark.sql.types.StructType = StructType(StructField(a,IntegerType,true))

scala> val asInt = expr("token = 'hello'").cast("int")
asInt: org.apache.spark.sql.Column = CAST((token = hello) AS INT)
```

When parsing, you should see INFO messages in the logs:

```
INFO CatalystSqlParser: Parsing command: int
```

It is also used in `HiveClientImpl` (when converting columns from Hive to Spark) and in `OrcFileOperator` (when inferring the schema for ORC files).

Tip

Enable `INFO` logging level for `org.apache.spark.sql.catalyst.parser.CatalystSqlParser` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.catalyst.parser.CatalystSqlParser=INFO
```

Refer to [Logging](#).

ParserInterface Contract — SQL Parsers

`ParserInterface` is the [abstraction](#) of [SQL parsers](#) that can convert ([parse](#)) textual representation of SQL statements into [Expressions](#), [LogicalPlans](#), [TableIdentifiers](#), [FunctionIdentifier](#), [StructType](#), and [DataType](#).

Table 1. ParserInterface Contract

Method	Description
<code>parseDataType</code>	<pre>parseDataType(sqlText: String): DataType</pre> <p>Parses a SQL text to an DataType</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataType</code> utility is requested to convert a DDL into a DataType (<code>DataType.fromDDL</code>) • <code>StructType</code> is requested to add a field • Column.cast • <code>HiveClientImpl</code> utility is requested to getSparkSQLDataType • <code>OrcFileOperator</code> is requested to <code>readSchema</code> • <code>PythonSQLUtils</code> is requested to <code>parseDataType</code> • <code>SQLUtils</code> is requested to <code>createStructField</code> • <code>OrcUtils</code> is requested to <code>readSchema</code>
<code>parseExpression</code>	<pre>parseExpression(sqlText: String): Expression</pre> <p>Parses a SQL text to an Expression</p> <p>Used in the following:</p> <ul style="list-style-type: none"> • Dataset operators: <code>Dataset.selectExpr</code>, <code>Dataset.filter</code> and <code>Dataset.where</code> • <code>expr</code> standard function
	<pre>parseFunctionIdentifier(sqlText: String): FunctionIdentifier</pre> <p>Parses a SQL text to an <code>FunctionIdentifier</code></p>

<code>parseFunctionIdentifier</code>	<p>Used when:</p> <ul style="list-style-type: none"> • <code>SessionCatalog</code> is requested to listFunctions • <code>CatalogImpl</code> is requested to getFunction and functionExists
<code>parsePlan</code>	<pre>parsePlan(sqlText: String): LogicalPlan</pre> <p>Parses a SQL text to a LogicalPlan</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>SessionCatalog</code> is requested to look up a relation (table or view) in catalogs • <code>SparkSession</code> is requested to execute a SQL query (aka SQL Mode)
<code>parseTableIdentifier</code>	<pre>parseTableIdentifier(sqlText: String): TableIdentifier</pre> <p>Parses a SQL text to a TableIdentifier</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataFrameWriter</code> is requested to insertInto and saveAsTable • <code>Dataset</code> is requested to createTempViewCommand • <code>SparkSession</code> is requested to table • <code>CatalogImpl</code> is requested to listColumns, getTable, tableExists, createTable, recoverPartitions, uncacheTable, and refreshTable • <code>SessionState</code> is requested to refreshTable
<code>parseTableSchema</code>	<pre>parseTableSchema(sqlText: String): StructType</pre> <p>Parses a SQL text to a schema (StructType)</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataType</code> utility is requested to convert a DDL into a DataType (DataType.fromDDL) • <code>StructType</code> utility is requested to create a StructType for a given DDL-formatted string (StructType.fromDDL)

- `JdbcUtils` utility is requested to `parseUserSpecifiedCreateTableColumnTypes` and `getCustomSchema`

Note

`AbstractSqlParser` is the base extension of the `ParserInterface` contract in Spark SQL.

`ParserInterface` is available as `sqlParser` property of `SessionState`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.sqlParser
org.apache.spark.sql.catalyst.parser.ParserInterface
```

SparkSqlAstBuilder

`SparkSqlAstBuilder` is an [AstBuilder](#) that converts valid Spark SQL statements into Catalyst expressions, logical plans or table identifiers (using [visit callback methods](#)).

Note

Spark SQL uses [ANTLR parser generator](#) for parsing structured text.

`SparkSqlAstBuilder` is created exclusively when `SparkSqlParser` is [created](#) (which is when `SparkSession` is requested for the lazily-created [SessionState](#)).

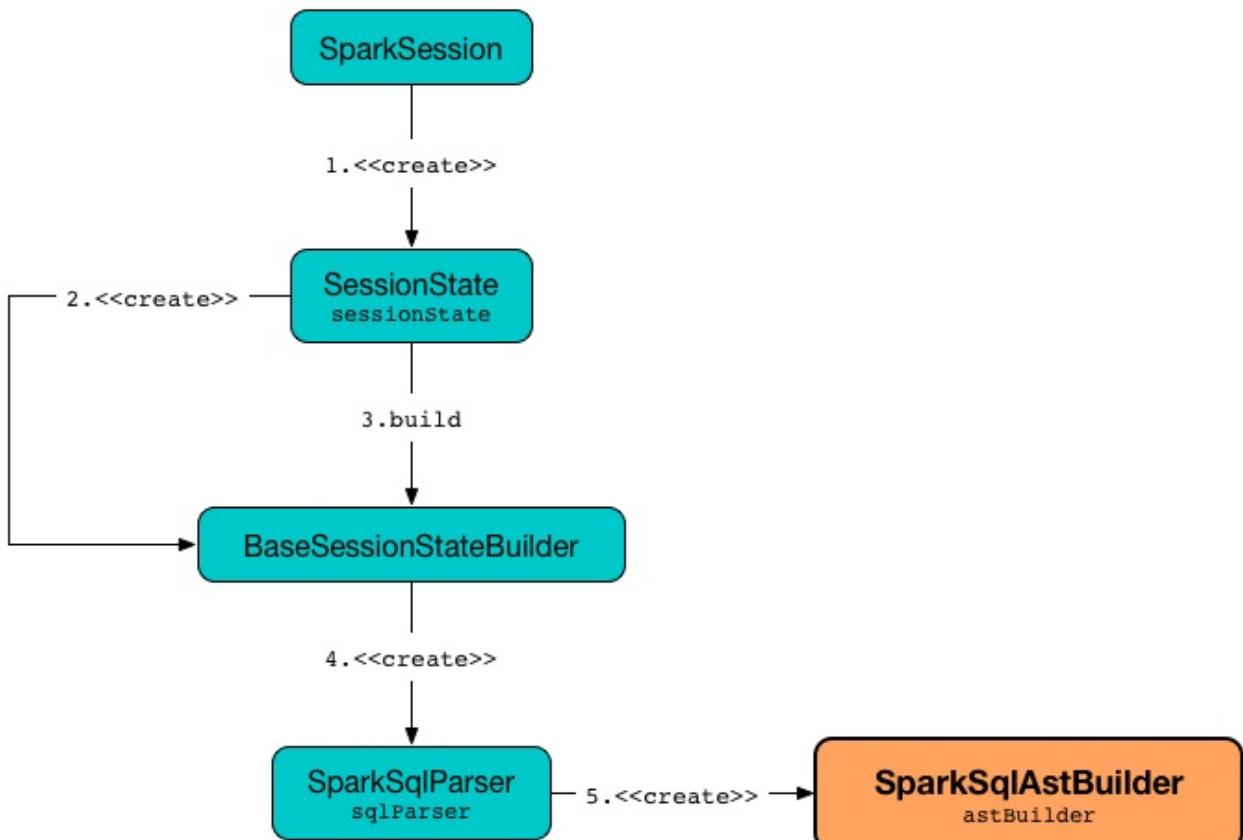


Figure 1. Creating `SparkSqlAstBuilder`

```

scala> :type spark.sessionState.sqlParser
org.apache.spark.sql.catalyst.parser.ParserInterface

import org.apache.spark.sql.execution.SparkSqlParser
val sqlParser = spark.sessionState.sqlParser.asInstanceOf[SparkSqlParser]

scala> :type sqlParser.astBuilder
org.apache.spark.sql.execution.SparkSqlAstBuilder
  
```

`SparkSqlAstBuilder` takes a [SQLConf](#) when created.

`SparkSqlAstBuilder` can also be temporarily created for `expr` standard function (to create column expressions).

Note

```
val c = expr("from_json(value, schema)")
scala> :type c
org.apache.spark.sql.Column

scala> :type c.expr
org.apache.spark.sql.catalyst.expressions.Expression

scala> println(c.expr.numberedTreeString)
00 'from_json('value, 'schema)
01 :- 'value
02 +- 'schema
```

Table 1. SparkSqlAstBuilder's Visit Callback

Callback Method	ANTLR rule / labeled alternative	
<code>visitAnalyze</code>	<code>#analyze</code>	<ul style="list-style-type: none"> • <code>AnalyzeColumnComma</code> <code>COLUMNS</code> clause (but no <code>FOR</code>) <pre>val sqlText = "ANALYZE val plan = spark.sql(import org.apache.spark.sql.AnalysisCommand) val cmd = plan.asInstanceOf[AnalyzeColumnCommand] scala> println(cmd) AnalyzeColumnCommand</pre> • <code>AnalyzePartitionComma</code> <code>specification</code> (but no <code>FOR</code>) <pre>val analyzeTable = "ANALYZE val plan = spark.sql(import org.apache.spark.sql.AnalysisCommand) val cmd = plan.asInstanceOf[AnalyzePartitionCommand] scala> println(cmd) AnalyzePartitionCommand</pre> • <code>AnalyzeTableComma</code> <code>PARTITION</code> specification <pre>val sqlText = "ANALYZE val plan = spark.sql(import org.apache.spark.sql.AnalysisCommand) val cmd = plan.asInstanceOf[AnalyzeTableCommand] scala> println(cmd) AnalyzeTableCommand</pre>

		<p>Note</p> <p>visitAnalyze <code>SyntaxException</code> if <code>NOSCAN</code> is used for logical commands</p>
<code>visitBucketSpec</code>	<code>#bucketSpec</code>	
<code>visitCacheTable</code>	<code>#cacheTable</code>	<ul style="list-style-type: none"> • <code>CacheTableCommand</code> <code>[query])?</code>
<code>visitCreateHiveTable</code>	<code>#createHiveTable</code>	<code>CreateTable</code>
<code>visitCreateTable</code>	<code>#createTable</code>	<ul style="list-style-type: none"> • <code>CreateTable</code> logical operations • <code>CreateTempViewUsing</code>
<code>visitCreateView</code>	<code>#createView</code>	<p><code>CreateViewCommand</code> for creating views</p> <pre>CREATE [OR REPLACE] [[GLOBAL] [TEMPORARY]] VIEW [IF NOT EXISTS] table_name [identifierCommentList] [[PARTITIONED BY column_name] [CLUSTERED BY column_name] [TBLPROPERTIES tableProperties]]</pre>
<code>visitCreateTempViewUsing</code>	<code>#createTempViewUsing</code>	<code>CreateTempViewUsing</code> for creating temporary views
		<ul style="list-style-type: none"> • <code>DescribeColumnCommand</code> for describing a column only (i.e. no <code>TABLE</code>) <pre>val sqlCmd = "DESC EXPLAIN t1" val plan = spark.sql(sqlCmd) import org.apache.spark.sql.catalyst.expressions._ val cmd = plan.asInstanceOf[DescribeColumnCommand] scala> println(cmd)</pre>
<code>visitDescribeTable</code>	<code>#describeTable</code>	<ul style="list-style-type: none"> • <code>DescribeTableCommand</code> for describing a <code>TABLE</code> (i.e. no column) <pre>val sqlCmd = "DESC t1" val plan = spark.sql(sqlCmd) import org.apache.spark.sql.catalyst.expressions._ val cmd = plan.asInstanceOf[DescribeTableCommand] scala> println(cmd)</pre>
<code>visitInsertOverwriteHiveDir</code>	<code>#insertOverwriteHiveDir</code>	

visitShowCreateTable	#showCreateTable	ShowCreateTableCommand SHOW CREATE TABLE tableId
----------------------	------------------	---

Table 2. SparkSqlAstBuilder's Parsing Handlers

Parsing Handler	LogicalPlan Added
withRepartitionByExpression	

SparkSqlParser — Default SQL Parser

`SparkSqlParser` is the default [SQL parser](#) of the SQL statements supported in Spark SQL.

`SparkSqlParser` supports [variable substitution](#).

`SparkSqlParser` uses [SparkSqlAstBuilder](#) (as `AstBuilder`).

Note	Spark SQL supports SQL statements as described in SqlBase.g4 ANTLR grammar.
------	---

`SparkSqlParser` is available as `sqlParser` of a `SessionState`.

```
val spark: SparkSession = ...
spark.sessionState.sqlParser
```

`SparkSqlParser` is used to translate an expression to the corresponding [Column](#) in the following:

- [expr](#) function
- [Dataset.selectExpr](#) operator
- [Dataset.filter](#) operator
- [Dataset.where](#) operator

```
scala> expr("token = 'hello'")
16/07/07 18:32:53 INFO SparkSqlParser: Parsing command: token = 'hello'
res0: org.apache.spark.sql.Column = (token = hello)
```

`SparkSqlParser` is used to parse table strings into their corresponding table identifiers in the following:

- `table` methods in [DataFrameReader](#) and [SparkSession](#)
- [insertInto](#) and [saveAsTable](#) methods of `DataFrameWriter`
- [createExternalTable](#) and [refreshTable](#) methods of [Catalog](#) (and [SessionState](#))

`SparkSqlParser` is used to translate a SQL text to its corresponding [logical operator](#) in [SparkSession.sql](#) method.

Enable `INFO` logging level for `org.apache.spark.sql.execution.SparkSqlParser` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

Tip

```
log4j.logger.org.apache.spark.sql.execution.SparkSqlParser=INFO
```

Refer to [Logging](#).

Variable Substitution

Caution

FIXME See `SparkSqlParser` and `substitutor` .

Thrift JDBC/ODBC Server — Spark Thrift Server (STS)

Thrift JDBC/ODBC Server (aka *Spark Thrift Server* or *STS*) is Spark SQL's port of [Apache Hive's HiveServer2](#) that allows JDBC/ODBC clients to execute SQL queries over JDBC and ODBC protocols on Apache Spark.

With Spark Thrift Server, business users can work with their shiny Business Intelligence (BI) tools, e.g. [Tableau](#) or Microsoft Excel, and connect to Apache Spark using the ODBC interface. That brings the in-memory distributed capabilities of Spark SQL's query engine (with all the [Catalyst query optimizations](#) you surely like very much) to environments that were initially "disconnected".

Beside, SQL queries in Spark Thrift Server share the same [SparkContext](#) that helps further improve performance of SQL queries using the same data sources.

Spark Thrift Server is a Spark standalone application that you start using [start-thriftserver.sh](#) and stop using [stop-thriftserver.sh](#) shell scripts.

Spark Thrift Server has its own tab in web UI — [JDBC/ODBC Server](#) available at [/sqlserver](#) URL.



Figure 1. Spark Thrift Server's web UI

Spark Thrift Server can work in [HTTP or binary transport modes](#).

Use [beeline command-line tool](#) or [SQirreL SQL Client](#) or Spark SQL's [DataSource API](#) to connect to Spark Thrift Server through the JDBC interface.

Spark Thrift Server extends [spark-submit](#)'s command-line options with [--hiveconf \[prop=value\]](#).

	You have to enable hive-thriftserver build profile to include Spark Thrift Server build.
Important	<pre>./build/mvn -Phadoop-2.7,yarn,mesos,hive,hive-thriftserver -DskipTests clean</pre> Refer to Building Apache Spark from Sources .

Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.sql.hive.thriftserver` and `org.apache.hive.service.server` loggers to see what happens inside.

Add the following line to `conf/log4j.properties` :

Tip

```
log4j.logger.org.apache.spark.sql.hive.thriftserver=DEBUG
log4j.logger.org.apache.hive.service.server=INFO
```

Refer to [Logging](#).

Starting Thrift JDBC/ODBC Server— `start-thriftserver.sh`

You can start Thrift JDBC/ODBC Server using `./sbin/start-thriftserver.sh` shell script.

With `INFO` logging level enabled, when you execute the script you should see the following `INFO` messages in the logs:

```
INFO HiveThriftServer2: Started daemon with process name: 16633@japila.local
INFO HiveThriftServer2: Starting SparkContext
...
INFO HiveThriftServer2: HiveThriftServer2 started
```

Internally, `start-thriftserver.sh` script submits

`org.apache.spark.sql.hive.thriftserver.HiveThriftServer2` standalone application for execution (using [spark-submit](#)).

```
$ ./bin/spark-submit --class org.apache.spark.sql.hive.thriftserver.HiveThriftServer2
```

Tip

Using the more explicit approach with `spark-submit` to start Spark Thrift Server could be easier to trace execution by seeing the logs printed out to the standard output and hence terminal directly.

Using Beeline JDBC Client to Connect to Spark Thrift Server

`beeline` is a command-line tool that allows you to access Spark Thrift Server using the JDBC interface on command line. It is included in the Spark distribution in `bin` directory.

```
$ ./bin/beeline
Beeline version 1.2.1.spark2 by Apache Hive
beeline>
```

You can connect to Spark Thrift Server using `connect` command as follows:

```
beeline> !connect jdbc:hive2://localhost:10000
```

When connecting in non-secure mode, simply enter the username on your machine and a blank password.

```
beeline> !connect jdbc:hive2://localhost:10000
Connecting to jdbc:hive2://localhost:10000
Enter username for jdbc:hive2://localhost:10000: jacek
Enter password for jdbc:hive2://localhost:10000: [press ENTER]
Connected to: Spark SQL (version 2.3.0)
Driver: Hive JDBC (version 1.2.1.spark2)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10000>
```

Once connected, you can send SQL queries (as if Spark SQL were a JDBC-compliant database).

```
0: jdbc:hive2://localhost:10000> show databases;
+-----+---+
| databaseName |
+-----+---+
| default     |
+-----+---+
1 row selected (0.074 seconds)
```

Connecting to Spark Thrift Server using SQuirreL SQL Client 3.7.1

Spark Thrift Server allows for remote access to Spark SQL using JDBC protocol.

Note	This section was tested with SQuirreL SQL Client 3.7.1 (<code>squirrelsql-3.7.1-standard.zip</code>) on Mac OS X.
------	---

SQuirreL SQL Client is a Java SQL client for JDBC-compliant databases.

Run the client using `java -jar squirrel-sql.jar`.

Figure 2. SQuirreL SQL Client

You first have to configure a JDBC driver for Spark Thrift Server. Spark Thrift Server uses `org.spark-project.hive:hive-jdbc:1.2.1.spark2` dependency that is the JDBC driver (that also downloads transitive dependencies).

Tip	The Hive JDBC Driver, i.e. <code>hive-jdbc-1.2.1.spark2.jar</code> and other jar files are in <code>jars</code> directory of the Apache Spark distribution (or <code>assembly/target/scala-2.11/jars</code> for local builds).
-----	--

Table 1. SQuirreL SQL Client's Connection Parameters

Parameter	Description
Name	Spark Thrift Server
Example URL	<code>jdbc:hive2://localhost:10000</code>
Extra Class Path	All the jar files of your Spark distribution
Class Name	<code>org.apache.hive.jdbc.HiveDriver</code>

Figure 3. Adding Hive JDBC Driver in SQuirreL SQL Client

With the Hive JDBC Driver defined, you can connect to Spark SQL Thrift Server.

Figure 4. Adding Hive JDBC Driver in SQuirreL SQL Client

Since you did not specify the database to use, Spark SQL's `default` is used.

Figure 5. SQuirreL SQL Client Connected to Spark Thrift Server (Metadata Tab)

Below is `show tables` SQL query in SQuirrel SQL Client executed in Spark SQL through Spark Thrift Server.

Figure 6. `show tables` SQL Query in SQuirrel SQL Client using Spark Thrift Server

Using Spark SQL's DataSource API to Connect to Spark Thrift Server

What might seem a quite artificial setup at first is accessing Spark Thrift Server using Spark SQL's [DataSource API](#), i.e. `DataFrameReader`'s [jdbc method](#).

When executed in `local` mode, Spark Thrift Server and `spark-shell` will try to access the same Hive Warehouse's directory that will inevitably lead to an error.

Use `spark.sql.warehouse.dir` to point to another directory for `spark-shell`.

Tip

```
./bin/spark-shell --conf spark.sql.warehouse.dir=/tmp/spark-warehouse
```

You should also not share the same home directory between them since `metastore_db` becomes an issue.

```
// Inside spark-shell
// Paste in :paste mode
val df = spark
  .read
  .option("url", "jdbc:hive2://localhost:10000") (1)
  .option("dbtable", "people") (2)
  .format("jdbc")
  .load
```

1. Connect to Spark Thrift Server at localhost on port 10000
2. Use `people` table. It assumes that `people` table is available.

ThriftServerTab — web UI's Tab for Spark Thrift Server

ThriftServerTab is...FIXME

Caution

FIXME Elaborate

Stopping Thrift JDBC/ODBC Server— `stop-thriftserver.sh`

You can stop a running instance of Thrift JDBC/ODBC Server using `./sbin/stop-thriftserver.sh` shell script.

With `DEBUG` logging level enabled, you should see the following messages in the logs:

```
ERROR HiveThriftServer2: RECEIVED SIGNAL TERM
DEBUG SparkSQLEnv: Shutting down Spark SQL Environment
INFO HiveServer2: Shutting down HiveServer2
INFO BlockManager: BlockManager stopped
INFO SparkContext: Successfully stopped SparkContext
```

Tip

You can also send `SIGTERM` signal to the process of Thrift JDBC/ODBC Server, i.e. `kill [PID]` that triggers the same sequence of shutdown steps as `stop-thriftserver.sh`.

Transport Mode

Spark Thrift Server can be configured to listen in two modes (aka *transport modes*):

1. **Binary mode** — clients should send thrift requests in binary
2. **HTTP mode** — clients send thrift requests over HTTP.

You can control the transport modes using `HIVE_SERVER2_TRANSPORT_MODE=http` or `hive.server2.transport.mode` (default: `binary`). It can be `binary` (default) or `http`.

main method

Thrift JDBC/ODBC Server is a Spark standalone application that you...

Caution	FIXME
---------	-------

HiveThriftServer2Listener

Caution	FIXME
---------	-------

SparkSQLEnv

Caution	FIXME
---------	-------

SQLExecution Helper Object

`SQLExecution` defines `spark.sql.execution.id` Spark property that is used to track multiple Spark jobs that should all together constitute a single structured query execution (that could be easily reported as a single execution unit).

```
import org.apache.spark.sql.execution.SQLExecution
scala> println(SQLExecution.EXECUTION_ID_KEY)
spark.sql.execution.id
```

Actions of a structured query are executed using `SQLExecution.withNewExecutionId` static method that sets `spark.sql.execution.id` as Spark Core's `local` property and "stitches" different Spark jobs as parts of one structured query action (that you can then see in web UI's [SQL tab](#)).

Use [SparkListener](#) to listen to [SparkListenerSQLExecutionStart](#) events and know the query executed in a Spark SQL application.

```
// "SQLAppStatusListener" idea is borrowed from
// Spark SQL's org.apache.spark.sql.execution.ui.SQLAppStatusListener
import org.apache.spark.scheduler.{SparkListener, SparkListenerEvent}
import org.apache.spark.sql.execution.ui.{SparkListenerDriverAccumUpdates, SparkListenerSQLExecutionStart, SparkListenerSQLExecutionEnd}
public class SQLAppStatusListener extends SparkListener {
    override def onOtherEvent(event: SparkListenerEvent): Unit = event match {
        case e: SparkListenerSQLExecutionStart => onExecutionStart(e)
        case e: SparkListenerSQLExecutionEnd => onExecutionEnd(e)
        case e: SparkListenerDriverAccumUpdates => onDriverAccumUpdates(e)
        case _ => // Ignore
    }
    def onExecutionStart(event: SparkListenerSQLExecutionStart): Unit = {
        // Find the QueryExecution for the Dataset action that triggered the event
        // This is the SQL-specific way
        import org.apache.spark.sql.execution.SQLExecution
        queryExecution = SQLExecution.getQueryExecution(event.executionId)
    }
    def onJobStart(jobStart: SparkListenerJobStart): Unit = {
        // Find the QueryExecution for the Dataset action that triggered the event
        // This is a general Spark Core way using local properties
        import org.apache.spark.sql.execution.SQLExecution
        val executionIdStr = jobStart.properties.getProperty(SQLExecution.EXECUTION_ID)
        // Note that the Spark job may or may not be a part of a structured query
        if (executionIdStr != null) {
            queryExecution = SQLExecution.getQueryExecution(executionIdStr.toLong)
        }
    }
    def onExecutionEnd(event: SparkListenerSQLExecutionEnd): Unit = {}
    def onDriverAccumUpdates(event: SparkListenerDriverAccumUpdates): Unit = {}
}

val sqlListener = new SQLAppStatusListener()
spark.sparkContext.addSparkListener(sqlListener)
```

Tip

Note

Jobs without `spark.sql.execution.id` key are not considered to belong to SQL query executions.

`SQLExecution` keeps track of all execution ids and their `QueryExecutions` in `executionIdToQueryExecution` internal registry.

Tip

Use `SQLExecution.getQueryExecution` to find the `QueryExecution` for an execution id.

Executing Dataset Action (with Zero or More Spark Jobs) Under New Execution Id — `withNewExecutionId` Method

```
withNewExecutionId[T](
    sparkSession: SparkSession,
    queryExecution: QueryExecution)(body: => T): T
```

`withNewExecutionId` executes `body` query action with a new `execution id` (given as the input `executionId` or auto-generated) so that all Spark jobs that have been scheduled by the query action could be marked as parts of the same `Dataset` action execution.

`withNewExecutionId` allows for collecting all the Spark jobs (even executed on separate threads) together under a single SQL query execution for reporting purposes, e.g. to reporting them as one single structured query in web UI.

Note

If there is another execution id already set, it is replaced for the course of the current action.

In addition, the `QueryExecution` variant posts `SparkListenerSQLExecutionStart` and `SparkListenerSQLExecutionEnd` events (to `LiveListenerBus` event bus) before and after executing the `body` action, respectively. It is used to inform `SQLListener` when a SQL query execution starts and ends.

Note

Nested execution ids are not supported in the `QueryExecution` variant.

Note

- `withNewExecutionId` is used when:
- `Dataset` is requested to `Dataset.withNewExecutionId` and `withAction`
 - `DataFrameWriter` is requested to `run a command`
 - Spark Structured Streaming's `streamExecution` commits a batch to a streaming sink
 - Spark Thrift Server's `SparksQLDriver` runs a command

Finding QueryExecution for Execution ID — `getQueryExecution` Method

```
getQueryExecution(executionId: Long): QueryExecution
```

`getQueryExecution` simply gives the `QueryExecution` for the `executionId` or `null` if not found.

Executing Action (with Zero or More Spark Jobs) Tracked Under Given Execution Id — `withExecutionId` Method

```
withExecutionId[T](
  sc: SparkContext,
  executionId: String)(body: => T): T
```

`withExecutionId` executes the `body` action as part of executing multiple Spark jobs under `executionId` `execution identifier`.

```
def body = println("Hello World")
scala> SQLExecution.withExecutionId(sc = spark.sparkContext, executionId = "Custom Name")(body)
Hello World
```

Note

- `withExecutionId` is used when:
- `BroadcastExchangeExec` is requested to `prepare for execution` (and initializes `relationFuture` for the first time)
 - `SubqueryExec` is requested to `prepare for execution` (and initializes `relationFuture` for the first time)

checkSQLExecutionId Method

```
checkSQLExecutionId(sparkSession: SparkSession): Unit
```

```
checkSQLExecutionId ...FIXME
```

Note

`checkSQLExecutionId` is used exclusively when `FileFormatWriter` is requested to write the result of a structured query.

withSQLConfPropagated Method

```
withSQLConfPropagated[T](sparkSession: SparkSession)(body: => T): T
```

```
withSQLConfPropagated ...FIXME
```

Note

`withSQLConfPropagated` is used when:

- `SQLExecution` is requested to `withNewExecutionId` and `withExecutionId`
- `TextInputJsonDataSource` is requested to `inferFromDataset`
- `MultiLineJsonDataSource` is requested to `infer`

RDDConversions Helper Object

`RDDConversions` is a Scala object that is used to `productToRowRdd` and `rowToRowRdd` methods.

productToRowRdd Method

```
productToRowRdd[A <: Product](data: RDD[A], outputTypes: Seq[DataType]): RDD[InternalRow]
```

`productToRowRdd` ...FIXME

Note

`productToRowRdd` is used when...FIXME

Converting Scala Objects In Rows to Values Of Catalyst Types — rowToRowRdd Method

```
rowToRowRdd(data: RDD[Row], outputTypes: Seq[DataType]): RDD[InternalRow]
```

`rowToRowRdd` maps over partitions of the input `RDD[Row]` (using `RDD.mapPartitions` operator) that creates a `MapPartitionsRDD` with a "map" function.

Tip

Use `RDD.toDebugString` to see the additional `MapPartitionsRDD` in an RDD lineage.

The "map" function takes a Scala `Iterator` of `Row` objects and does the following:

1. Creates a `GenericInternalRow` (of the size that is the number of columns per the input `Seq[DataType]`)
2. Creates a converter function for every `DataType` in `Seq[DataType]`
3. For every `Row` object in the partition (iterator), applies the converter function per position and adds the result value to the `GenericInternalRow`
4. In the end, returns a `GenericInternalRow` for every row

Note

`rowToRowRdd` is used exclusively when `DataSourceStrategy` execution planning strategy is `executed` (and requested to `toCatalystRDD`).

CatalystTypeConverters Helper Object

`CatalystTypeConverters` is a Scala object that is used to convert Scala types to Catalyst types and vice versa.

createToCatalystConverter Method

```
createToCatalystConverter(dataType: DataType): Any => Any
```

`createToCatalystConverter` ...FIXME

Note	<code>createToCatalystConverter</code> is used when...FIXME
------	---

convertToCatalyst Method

```
convertToCatalyst(a: Any): Any
```

`convertToCatalyst` ...FIXME

Note	<code>convertToCatalyst</code> is used when...FIXME
------	---

StatFunctions Helper Object

`StatFunctions` is a Scala object that defines the [methods](#) that are used for...FIXME

Table 1. StatFunctions API

Method	Description
calculateCov	<code>calculateCov(df: DataFrame, cols: Seq[String]): Double</code>
crossTabulate	<code>crossTabulate(df: DataFrame, col1: String, col2: String): D</code>
multipleApproxQuantiles	<code>multipleApproxQuantiles(df: DataFrame, cols: Seq[String], probabilities: Seq[Double], relativeError: Double): Seq[Seq[Double]]</code>
pearsonCorrelation	<code>pearsonCorrelation(df: DataFrame, cols: Seq[String]): Doub</code>
summary	<code>summary(ds: Dataset[_], statistics: Seq[String]): DataFrame</code>

calculateCov Method

```
calculateCov(df: DataFrame, cols: Seq[String]): Double
```

`calculateCov` ...FIXME

Note

`calculateCov` is used when...FIXME

crossTabulate Method

```
crossTabulate(  
  df: DataFrame,  
  col1: String,  
  col2: String): DataFrame
```

```
crossTabulate ...FIXME
```

Note

`crossTabulate` is used when...FIXME

multipleApproxQuantiles Method

```
multipleApproxQuantiles(  
  df: DataFrame,  
  cols: Seq[String],  
  probabilities: Seq[Double],  
  relativeError: Double): Seq[Seq[Double]]
```

```
multipleApproxQuantiles ...FIXME
```

Note

`multipleApproxQuantiles` is used when...FIXME

pearsonCorrelation Method

```
pearsonCorrelation(df: DataFrame, cols: Seq[String]): Double
```

```
pearsonCorrelation ...FIXME
```

Note

`pearsonCorrelation` is used when...FIXME

Generate Summary Statistics of Dataset (as DataFrame)

— summary Method

```
summary(  
  ds: Dataset[_],  
  statistics: Seq[String]): DataFrame
```

```
summary ...FIXME
```

Note

`summary` is used exclusively when [Dataset.summary](#) action is used.

SubExprUtils Helper Object

`SubExprUtils` is a Scala object that is used for...FIXME

`SubExprUtils` uses [PredicateHelper](#) for...FIXME

`SubExprUtils` is used to [check whether a condition expression has any null-aware predicate subqueries inside Not expressions](#).

Checking If Condition Expression Has Any Null-Aware Predicate Subqueries Inside Not

— `hasNullAwarePredicateWithinNot` Method

```
hasNullAwarePredicateWithinNot(condition: Expression): Boolean
```

`hasNullAwarePredicateWithinNot` splits conjunctive predicates (i.e. expressions separated by `And` expression).

`hasNullAwarePredicateWithinNot` is positive (i.e. `true`) and is considered to have a **null-aware predicate subquery inside a Not expression** when conjunctive predicate expressions include a `Not` expression with an `In` predicate expression with a `ListQuery` subquery expression.

```
import org.apache.spark.sql.catalyst.plans.logical._
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = LocalRelation('key.int, 'value.string).analyze

import org.apache.spark.sql.catalyst.expressions._
val in = In(value = Literal.create(1), Seq(ListQuery(plan)))
val condition = Not(child = Or(left = Literal.create(false), right = in))

import org.apache.spark.sql.catalyst.expressions.SubExprUtils
val positive = SubExprUtils.hasNullAwarePredicateWithinNot(condition)
assert(positive)
```

`hasNullAwarePredicateWithinNot` is negative (i.e. `false`) for all the other expressions and in particular the following expressions:

1. [Exists](#) predicate subquery expressions
2. `Not` expressions with a [Exists](#) predicate subquery expression as the child expression
3. `In` expressions with a [ListQuery](#) subquery expression as the `list` expression

4. Not expressions with a In expression (with a ListQuery subquery expression as the list expression)

```

import org.apache.spark.sql.catalyst.plans.logical._
import org.apache.spark.sql.catalyst.dsl.plans._
val plan = LocalRelation('key.int, 'value.string).analyze

import org.apache.spark.sql.catalyst.expressions._
import org.apache.spark.sql.catalyst.expressions.SubExprUtils

// Exists
val condition = Exists(plan)
val negative = SubExprUtils.hasNullAwarePredicateWithinNot(condition)
assert(!negative)

// Not Exists
val condition = Not(child = Exists(plan))
val negative = SubExprUtils.hasNullAwarePredicateWithinNot(condition)
assert(!negative)

// In with ListQuery
val condition = In(value = Literal.create(1), Seq(ListQuery(plan)))
val negative = SubExprUtils.hasNullAwarePredicateWithinNot(condition)
assert(!negative)

// Not In with ListQuery
val in = In(value = Literal.create(1), Seq(ListQuery(plan)))
val condition = Not(child = in)
val negative = SubExprUtils.hasNullAwarePredicateWithinNot(condition)
assert(!negative)

```

Note

`hasNullAwarePredicateWithinNot` is used exclusively when `CheckAnalysis` analysis validation is requested to validate analysis of a logical plan (with `Filter` logical operators).

PredicateHelper Scala Trait

`PredicateHelper` defines the [methods](#) that are used to work with predicates (mainly).

Table 1. PredicateHelper's Methods

Method	Description
splitConjunctivePredicates	
splitDisjunctivePredicates	
replaceAlias	
canEvaluate	
canEvaluateWithinJoin	

Splitting Conjunctive Predicates

— `splitConjunctivePredicates` Method

```
splitConjunctivePredicates(condition: Expression): Seq[Expression]
```

`splitConjunctivePredicates` takes the input condition [expression](#) and splits it to two expressions if they are children of a `And` binary expression.

`splitConjunctivePredicates` splits the child expressions recursively down the child expressions until no conjunctive `And` binary expressions exist.

splitDisjunctivePredicates Method

```
splitDisjunctivePredicates(condition: Expression): Seq[Expression]
```

`splitDisjunctivePredicates` ...FIXME

Note	<code>splitDisjunctivePredicates</code> is used when...FIXME
------	--

replaceAlias Method

```
replaceAlias(  
    condition: Expression,  
    aliases: AttributeMap[Expression]): Expression
```

`replaceAlias ...FIXME`

Note

`replaceAlias` is used when...FIXME

canEvaluate Method

```
canEvaluate(expr: Expression, plan: LogicalPlan): Boolean
```

`canEvaluate ...FIXME`

Note

`canEvaluate` is used when...FIXME

canEvaluateWithinJoin Method

```
canEvaluateWithinJoin(expr: Expression): Boolean
```

`canEvaluateWithinJoin` indicates whether a [Catalyst expression](#) *can be evaluated within a join*, i.e. when one of the following conditions holds:

- Expression is [deterministic](#)
- Expression is not `Unevaluable`, `ListQuery` or `Exists`
- Expression is a `SubqueryExpression` with no child expressions
- Expression is a `AttributeReference`
- Any expression with child expressions that meet one of the above conditions

Note

`canEvaluateWithinJoin` is used when:

- `PushPredicateThroughJoin` logical optimization rule is [executed](#)
- `ReorderJoin` logical optimization rule does [createOrderedJoin](#)

SchemaUtils Helper Object

`SchemaUtils` is a Scala object that is used for the following:

- [checkColumnNameDuplication](#)
- [checkSchemaColumnNameDuplication](#)

checkColumnNameDuplication Method

```
checkColumnNameDuplication(  
    columnNames: Seq[String], colType: String, resolver: Resolver): Unit (1)  
checkColumnNameDuplication(  
    columnNames: Seq[String], colType: String, caseSensitiveAnalysis: Boolean): Unit
```

1. Uses the other `checkColumnNameDuplication` with `caseSensitiveAnalysis` flag per [isCaseSensitiveAnalysis](#)

`checkColumnNameDuplication ...FIXME`

Note

`checkColumnNameDuplication` is used when...FIXME

checkSchemaColumnNameDuplication Method

```
checkSchemaColumnNameDuplication(  
    schema: StructType, colType: String, caseSensitiveAnalysis: Boolean = false): Unit
```

`checkSchemaColumnNameDuplication ...FIXME`

Note

`checkSchemaColumnNameDuplication` is used when...FIXME

isCaseSensitiveAnalysis Internal Method

```
isCaseSensitiveAnalysis(resolver: Resolver): Boolean
```

`isCaseSensitiveAnalysis ...FIXME`

Note

`isCaseSensitiveAnalysis` is used when...FIXME

AggUtils Helper Object

`AggUtils` is a Scala object that defines the methods used exclusively when [Aggregation](#) execution planning strategy is executed.

- [planAggregateWithoutDistinct](#)
- [planAggregateWithOneDistinct](#)

planAggregateWithOneDistinct Method

```
planAggregateWithOneDistinct(
    groupingExpressions: Seq[NamedExpression],
    functionsWithDistinct: Seq[AggregateExpression],
    functionsWithoutDistinct: Seq[AggregateExpression],
    resultExpressions: Seq[NamedExpression],
    child: SparkPlan): Seq[SparkPlan]
```

`planAggregateWithOneDistinct ...FIXME`

Note	<code>planAggregateWithOneDistinct</code> is used exclusively when Aggregation execution planning strategy is executed .
------	--

Creating Physical Plan with Two Aggregate Physical Operators for Partial and Final Aggregations

— planAggregateWithoutDistinct Method

```
planAggregateWithoutDistinct(
    groupingExpressions: Seq[NamedExpression],
    aggregateExpressions: Seq[AggregateExpression],
    resultExpressions: Seq[NamedExpression],
    child: SparkPlan): Seq[SparkPlan]
```

`planAggregateWithoutDistinct` is a two-step physical operator generator.

`planAggregateWithoutDistinct` first [creates an aggregate physical operator](#) with `aggregateExpressions` in `Partial` mode (for partial aggregations).

Note	<code>requiredChildDistributionExpressions</code> for the aggregate physical operator for partial aggregation "stage" is empty.
------	---

In the end, `planAggregateWithoutDistinct` creates another aggregate physical operator (of the same type as before), but `aggregateExpressions` are now in `Final` mode (for final aggregations). The aggregate physical operator becomes the parent of the first aggregate operator.

Note	<code>requiredChildDistributionExpressions</code> for the parent aggregate physical operator for final aggregation "stage" are the <code>attributes</code> of <code>groupingExpressions</code> .
------	--

Note	<code>planAggregateWithoutDistinct</code> is used exclusively when Aggregation execution planning strategy is <code>executed</code> (with no <code>AggregateExpressions</code> being <code>distinct</code>).
------	---

Creating Aggregate Physical Operator — `createAggregate` Internal Method

```
createAggregate(  
    requiredChildDistributionExpressions: Option[Seq[Expression]] = None,  
    groupingExpressions: Seq[NamedExpression] = Nil,  
    aggregateExpressions: Seq[AggregateExpression] = Nil,  
    aggregateAttributes: Seq[Attribute] = Nil,  
    initialInputBufferOffset: Int = 0,  
    resultExpressions: Seq[NamedExpression] = Nil,  
    child: SparkPlan): SparkPlan
```

`createAggregate` creates a physical operator given the input `aggregateExpressions` aggregate expressions.

Table 1. `createAggregate`'s Aggregate Physical Operator Selection Criteria (in execution order)

Aggregate Physical Operator	Selection Criteria
<code>HashAggregateExec</code>	<code>HashAggregateExec</code> supports all <code>aggBufferAttributes</code> of the input <code>aggregateExpressions</code> aggregate expressions.
<code>ObjectHashAggregateExec</code>	<ol style="list-style-type: none"> <code>spark.sql.execution.useObjectHashAggregateExec</code> internal flag enabled (it is by default) <code>ObjectHashAggregateExec</code> supports the input <code>aggregateExpressions</code> aggregate expressions.
<code>SortAggregateExec</code>	When all the above requirements could not be met.

Note	<code>createAggregate</code> is used when <code>Aggutils</code> is requested to <code>planAggregateWithoutDistinct</code> , <code>planAggregateWithOneDistinct</code> (and <code>planStreamingAggregation</code> for Spark Structured Streaming)
------	--

ScalaReflection

`ScalaReflection` is the contract and the only implementation of the contract with...FIXME

serializerFor Object Method

```
serializerFor[T : TypeTag](inputObject: Expression): CreateNamedStruct
```

`serializerFor` firstly finds the [local type of](#) the input type `T` and then the [class name](#).

`serializerFor` uses the [internal version of itself](#) with the input `inputObject` expression, the `tpe` type and the `walkedTypePath` with the class name found earlier (of the input type `T`).

```
- root class: "[clsName]"
```

In the end, `serializerFor` returns one of the following:

- The [CreateNamedStruct](#) expression from the false value of the `If` expression returned only if the type `T` is [definedByConstructorParams](#)
- Creates a [CreateNamedStruct](#) expression with the [Literal](#) with the `value` as `"value"` and the expression returned

```
import org.apache.spark.sql.functions.lit
val inputObject = lit(1).expr

import org.apache.spark.sql.catalystScalaReflection
val serializer = ScalaReflection.serializerFor(inputObject)
scala> println(serializer)
named_struct(value, 1)
```

Note	<code>serializerFor</code> is used when...FIXME
------	---

serializerFor Internal Method

```
serializerFor(
  inputObject: Expression,
  tpe: `Type`,
  walkedTypePath: Seq[String],
  seenTypeSet: Set[`Type`] = Set.empty): Expression
```

`serializerFor ...FIXME`

Note	<code>serializerFor</code> is used exclusively when <code>ScalaReflection</code> is requested to <code>serializerFor</code> .
------	---

localTypeOf Object Method

`localTypeOf[T: TypeTag]: `Type``

`localTypeOf ...FIXME`

```
import org.apache.spark.sql.catalystScalaReflection
val tpe = ScalaReflection.localTypeOf[Int]
scala> :type tpe
org.apache.spark.sql.catalystScalaReflection.universe.Type

scala> println(tpe)
Int
```

Note	<code>localTypeOf</code> is used when...FIXME
------	---

getClassNameFromType Object Method

`getClassNameFromType(tpe: `Type`): String`

`getClassNameFromType ...FIXME`

```
import org.apache.spark.sql.catalystScalaReflection
val tpe = ScalaReflection.localTypeOf[java.time.LocalDateTime]
val className = ScalaReflection.getClassNameFromType(tpe)
scala> println(className)
java.time.LocalDateTime
```

Note	<code>getClassNameFromType</code> is used when...FIXME
------	--

definedByConstructorParams Object Method

`definedByConstructorParams(tpe: Type): Boolean`

`definedByConstructorParams ...FIXME`

Note

`definedByConstructorParams` is used when...FIXME

CreateStruct Function Builder

`CreateStruct` is a [function builder](#) (e.g. `Seq[Expression] → Expression`) that can [create](#) `CreateNamedStruct` [expressions](#) and is the [metadata](#) of the `struct` function.

Metadata of struct Function — `registryEntry` Property

```
registryEntry: (String, (ExpressionInfo, FunctionBuilder))
```

`registryEntry` ...FIXME

Note

`registryEntry` is used exclusively when `FunctionRegistry` is requested for the [function expression registry](#).

Creating CreateNamedStruct Expression — `apply` Method

```
apply(children: Seq[Expression]): CreateNamedStruct
```

Note

`apply` is part of Scala's [scala.Function1](#) contract to create a function of one parameter (e.g. `Seq[Expression]`).

`apply` creates a [CreateNamedStruct](#) expression with the input `children` [expressions](#) as follows:

- For [NamedExpression](#) expressions that are [resolved](#), `apply` creates a pair of a [Literal](#) expression (with the [name](#) of the `NamedExpression`) and the `NamedExpression` itself
- For [NamedExpression](#) expressions that are not [resolved](#) yet, `apply` creates a pair of a [NamePlaceholder](#) expression and the `NamedExpression` itself
- For all other [expressions](#), `apply` creates a pair of a [Literal](#) expression (with the value as `col[index]`) and the `Expression` itself

Note

- `apply` is used when:
- `ResolveReferences` logical resolution rule is requested to [expandStarExpression](#)
 - `InConversion` type coercion rule is requested to [coerceTypes](#)
 - `ExpressionEncoder` is requested to [create an ExpressionEncoder for a tuple](#)
 - `Stack` generator expression is requested to [generate a Java source code](#)
 - `AstBuilder` is requested to parse a [struct](#) and row constructor
 - `ColumnStat` is requested to [statExprs](#)
 - `KeyValueGroupedDataset` is requested to [aggUntyped](#) (when `KeyValueGroupedDataset.agg` typed operator is used)
 - `Dataset.joinWith` typed transformation is used
 - `struct` standard function is used
 - `SimpleTypedAggregateExpression` expression is requested for the [evaluateExpression](#) and [resultObjToRow](#)

MultiInstanceRelation

`MultiInstanceRelation` is a [contact](#) of [logical operators](#) which a [single instance](#) might appear multiple times in a logical query plan.

```
package org.apache.spark.sql.catalyst.analysis

trait MultiInstanceRelation {
  def newInstance(): LogicalPlan
}
```

When `ResolveReferences` logical evaluation is [executed](#), every `MultiInstanceRelation` in a logical query plan is requested to [produce a new version of itself with globally unique expression ids](#).

Table 1. MultiInstanceRelations

MultiInstanceRelation	Description
ContinuousExecutionRelation	Used in Spark Structured Streaming
DataSourceV2Relation	
ExternalRDD	
HiveTableRelation	
InMemoryRelation	
LocalRelation	
LogicalRDD	
LogicalRelation	
Range	
View	
StreamingExecutionRelation	Used in Spark Structured Streaming
StreamingRelation	Used in Spark Structured Streaming
StreamingRelationV2	Used in Spark Structured Streaming

TypeCoercion Object

`TypeCoercion` is a Scala object that defines the [type coercion rules](#) for [Spark Analyzer](#).

Defining Type Coercion Rules (For Spark Analyzer) — `typeCoercionRules` Method

```
typeCoercionRules(conf: SQLConf): List[Rule[LogicalPlan]]
```

`typeCoercionRules` is a collection of [Catalyst rules](#) to transform [logical plans](#) (in the order of execution):

1. [InConversion](#)
2. [WidenSetOperationTypes](#)
3. [PromoteStrings](#)
4. [DecimalPrecision](#)
5. [BooleanEquality](#)
6. [FunctionArgumentConversion](#)
7. [ConcatCoercion](#)
8. [EltCoercion](#)
9. [CaseWhenCoercion](#)
10. [IfCoercion](#)
11. [StackCoercion](#)
12. [Division](#)
13. [ImplicitTypeCasts](#)
14. [DateTimeOperations](#)
15. [WindowFrameCoercion](#)

Note	<code>typeCoercionRules</code> is used exclusively when <code>Analyzer</code> is requested for batches.
------	---

TypeCoercionRule Contract — Type Coercion Rules

`TypeCoercionRule` is the [contract](#) of logical rules to [coerce](#) and [propagate](#) types in [logical plans](#).

```
package org.apache.spark.sql.catalyst.analysis

trait TypeCoercionRule extends Rule[LogicalPlan] {
    // only required methods that have no implementation
    // the others follow
    def coerceTypes(plan: LogicalPlan): LogicalPlan
}
```

Table 1. (Subset of) TypeCoercionRule Contract

Method	Description
<code>coerceTypes</code>	Coerce types in a logical plan Used exclusively when <code>TypeCoercionRule</code> is executed

`TypeCoercionRule` is simply a [Catalyst rule](#) for transforming [logical plans](#), i.e. `Rule[LogicalPlan]`.

Table 2. TypeCoercionRules

TypeCoercionRule	Description
CaseWhenCoercion	
ConcatCoercion	
DecimalPrecision	
Division	
EltCoercion	
FunctionArgumentConversion	
IfCoercion	
ImplicitTypeCasts	
InConversion	
PromoteStrings	
StackCoercion	
WindowFrameCoercion	

Executing Rule — apply Method

```
apply(plan: LogicalPlan): LogicalPlan
```

Note

`apply` is part of the [Rule Contract](#) to execute (apply) a rule on a [TreeNode](#) (e.g. [LogicalPlan](#)).

`apply coerceTypes` in the input [LogicalPlan](#) and returns the following:

- The input [LogicalPlan](#) itself if [matches](#) the coerced plan
- The plan after [propagateTypes](#) on the coerced plan

propagateTypes Internal Method

```
propagateTypes(plan: LogicalPlan): LogicalPlan
```

propagateTypes ...FIXME

Note	propagateTypes is used exclusively when TypeCoercionRule is executed .
------	--

ExtractEquiJoinKeys — Scala Extractor for Destructuring Join Logical Operators

`ExtractEquiJoinKeys` is a Scala extractor to [destruct a Join logical operator](#) into a tuple with the following elements:

1. [Join type](#)
2. Left and right keys (for non-empty join keys in the [condition](#) of the `Join` operator)
3. Join condition (i.e. a Catalyst expression that could be used as a new join condition)
4. The [left](#) and the [right](#) logical operators

ReturnType

```
(JoinType, Seq[Expression], Seq[Expression], Option[Expression], LogicalPlan, LogicalPlan)
```

`unapply` gives `None` (aka *nothing*) when no join keys were found or the logical plan is not a Join logical operator.

Note	<code>ExtractEquiJoinKeys</code> is a Scala object with <code>unapply</code> method.
------	--

```

val left = Seq((0, 1, "zero"), (1, 2, "one")).toDF("k1", "k2", "name")
val right = Seq((0, 0, "0"), (1, 1, "1")).toDF("k1", "k2", "name")
val q = left.join(right, Seq("k1", "k2", "name")).
    where(left("k1") > 3)
import org.apache.spark.sql.catalyst.plans.logical.Join
val join = q.queryExecution.optimizedPlan.p(1).asInstanceOf[Join]

// make sure the join condition is available
scala> join.condition.get
res1: org.apache.spark.sql.catalyst.expressions.Expression = (((k1#148 = k1#161) && (k2#149 = k2#162)) && (name#150 = name#163))

// Enable DEBUG logging level
import org.apache.log4j.{Level, Logger}
Logger.getLogger("org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys").setLevel(Level.DEBUG)

import org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys
scala> ExtractEquiJoinKeys.unapply(join)
2018-03-14 12:02:14 DEBUG ExtractEquiJoinKeys:58 - Considering join on: Some(((k1#148 = k1#161) && (k2#149 = k2#162)) && (name#150 = name#163))
2018-03-14 12:02:14 DEBUG ExtractEquiJoinKeys:58 - leftKeys>List(k1#148, k2#149, name#150) | rightKeys>List(k1#161, k2#162, name#163)
res3: Option[org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys.ReturnType] = Some((Inner, List(k1#148, k2#149, name#150), List(k1#161, k2#162, name#163), None, Project
  [_1#144 AS k1#148, _2#145 AS k2#149, _3#146 AS name#150]
  +- Filter ((_1#144 > 3) && isnotnull(_3#146))
    +- LocalRelation [_1#144, _2#145, _3#146]
,Project [_1#157 AS k1#161, _2#158 AS k2#162, _3#159 AS name#163]
  +- Filter ((_1#157 > 3) && isnotnull(_3#159))
    +- LocalRelation [_1#157, _2#158, _3#159]
))

```

Tip

Enable `DEBUG` logging level for `org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.catalyst.planning.ExtractEquiJoinKeys=DEBUG
```

Refer to [Logging](#).

Destructuring Logical Plan — `unapply` Method

```

type ReturnType =
  (JoinType, Seq[Expression], Seq[Expression], Option[Expression], LogicalPlan, LogicalPlan)

unapply(plan: LogicalPlan): Option[ReturnType]

```

Internally, `unapply` prints out the following DEBUG message to the logs:

```
Considering join on: [condition]
```

`unapply` then splits `condition` at `And` expression points (if there are any) to have a list of predicate expressions.

Caution	FIXME Example with a condition with multiple predicates separated by ANDs.
---------	--

`unapply` finds `EqualTo` and `EqualNullSafe` binary expressions to collect the join keys (for the left and right side).

Caution	FIXME 5 examples for the different cases of <code>EqualTo</code> and <code>EqualNullSafe</code> binary expressions.
---------	---

`unapply` takes the expressions that...FIXME...to build `otherPredicates`.

In the end, `unapply` splits the pairs of join keys into collections of left and right join keys.

`unapply` prints out the following DEBUG message to the logs:

```
leftKeys:[leftKeys] | rightKeys:[rightKeys]
```

Note	<code>unapply</code> is used when: <ul style="list-style-type: none"> • <code>JoinEstimation</code> is requested to <code>estimateInnerOuterJoin</code> • <code>JoinSelection</code> execution planning strategy is <code>executed</code> • (Spark Structured Streaming) <code>StreamingJoinStrategy</code> execution planning strategy is executed • (Spark Structured Streaming) <code>StreamingJoinHelper</code> is requested to find the watermark in the join keys
------	---

PhysicalAggregation — Scala Extractor for Destructuring Aggregate Logical Operators

`PhysicalAggregation` is a Scala extractor to [destructre an Aggregate logical operator](#) into a four-element tuple with the following elements:

1. Grouping [named expressions](#)
2. [AggregateExpressions](#)
3. Result [named expressions](#)
4. Child [logical operator](#)

`ReturnType`

```
(Seq[NamedExpression], Seq[AggregateExpression], Seq[NamedExpression], LogicalPlan)
```

Tip

See the document about [Scala extractor objects](#).

Destructuring Aggregate Logical Operator — `unapply` Method

```
type ReturnType =
  (Seq[NamedExpression], Seq[AggregateExpression], Seq[NamedExpression], LogicalPlan)

unapply(a: Any): Option[ReturnType]
```

`unapply` destructures the input `a` [Aggregate logical operator](#) into a four-element `ReturnType`.

Note

`unapply` is used when...FIXME

PhysicalOperation — Scala Extractor for Destructuring Logical Query Plans

`PhysicalOperation` is a Scala extractor to [destruct a logical query plan](#) into a tuple with the following elements:

1. [Named expressions](#) (aka *projects*)
2. [Expressions](#) (aka *filters*)
3. [Logical operator](#) (aka *leaf node*)

`ReturnType`

```
(Seq[NamedExpression], Seq[Expression], LogicalPlan)
```

The following idiom is often used in `strategy` implementations (e.g. [HiveTableScans](#), [InMemoryScans](#), [DataSourceStrategy](#), [FileSourceStrategy](#)):

```
def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
  case PhysicalOperation(projections, predicates, plan) =>
    // do something
  case _ => Nil
}
```

Whenever used to pattern match to a `LogicalPlan`, `PhysicalOperation`'s `unapply` is called.

unapply Method

```
type ReturnType = (Seq[NamedExpression], Seq[Expression], LogicalPlan)

unapply(plan: LogicalPlan): Option[ReturnType]
```

`unapply ...FIXME`

Note

`unapply` is almost [collectProjectsAndFilters](#) method itself (with some manipulations of the return value).

Note

`unapply` is used when...FIXME

HashJoin — Contract for Hash-based Join Physical Operators

`HashJoin` is the [contract](#) for hash-based join physical operators (e.g. [BroadcastHashJoinExec](#) and [ShuffledHashJoinExec](#)).

```
package org.apache.spark.sql.execution.joins

trait HashJoin {
    // only required methods that have no implementation
    // the others follow
    val leftKeys: Seq[Expression]
    val rightKeys: Seq[Expression]
    val joinType: JoinType
    val buildSide: BuildSide
    val condition: Option[Expression]
    val left: SparkPlan
    val right: SparkPlan
}
```

Table 1. HashJoin Contract

Method	Description
<code>buildSide</code>	<p>Left or right build side</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>HashJoin</code> is requested for <code>buildPlan</code>, <code>streamedPlan</code>, <code>buildKeys</code> and <code>streamedKeys</code> • <code>BroadcastHashJoinExec</code> physical operator is requested for <code>requiredChildDistribution</code>, to <code>codegenInner</code> and <code>codegenOuter</code>
<code>joinType</code>	<code>JoinType</code>

Table 2. HashJoin's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
boundCondition	
buildKeys	Build join keys (as Catalyst expressions)
buildPlan	
streamedKeys	Streamed join keys (as Catalyst expressions)
streamedPlan	

join Method

```
join(
    streamedIter: Iterator[InternalRow],
    hashed: HashedRelation,
    numOutputRows: SQLMetric,
    avgHashProbe: SQLMetric): Iterator[InternalRow]
```

`join` branches off per [joinType](#) to create a join iterator of internal rows (i.e. `Iterator[InternalRow]`) for the input `streamedIter` and `hashed`:

- [innerJoin](#) for a [InnerLike](#) join
- [outerJoin](#) for a [LeftOuter](#) or a [RightOuter](#) join
- [semiJoin](#) for a [LeftSemi](#) join
- [antiJoin](#) for a [LeftAnti](#) join
- [existenceJoin](#) for a [ExistenceJoin](#) join

`join` requests `TaskContext` to add a `TaskCompletionListener` to update the input avg hash probe SQL metric. The `TaskCompletionListener` is executed on a task completion (regardless of the task status: success, failure, or cancellation) and uses `getAverageProbesPerLookup` from the input `hashed` to set the input avg hash probe.

`join` [createResultProjection](#).

In the end, for every row in the join iterator of internal rows `join` increments the input `numOutputRows` SQL metric and applies the result projection.

`join` reports a `IllegalArgumentException` when the `joinType` is incorrect.

[x] JoinType is not supported

Note	join is used when BroadcastHashJoinExec and ShuffledHashJoinExec are executed.
------	--

innerJoin Internal Method

```
innerJoin(  
    streamIter: Iterator[InternalRow],  
    hashedRelation: HashedRelation): Iterator[InternalRow]
```

innerJoin ...FIXME

Note	innerJoin is used when...FIXME
------	--------------------------------

outerJoin Internal Method

```
outerJoin(  
    streamedIter: Iterator[InternalRow],  
    hashedRelation: HashedRelation): Iterator[InternalRow]
```

outerJoin ...FIXME

Note	outerJoin is used when...FIXME
------	--------------------------------

semiJoin Internal Method

```
semiJoin(  
    streamIter: Iterator[InternalRow],  
    hashedRelation: HashedRelation): Iterator[InternalRow]
```

semiJoin ...FIXME

Note	semiJoin is used when...FIXME
------	-------------------------------

antiJoin Internal Method

```
antiJoin(  
    streamIter: Iterator[InternalRow],  
    hashedRelation: HashedRelation): Iterator[InternalRow]
```

antiJoin ...FIXME

Note

antiJoin is used when...FIXME

existenceJoin Internal Method

```
existenceJoin(  
    streamIter: Iterator[InternalRow],  
    hashedRelation: HashedRelation): Iterator[InternalRow]
```

existenceJoin ...FIXME

Note

existenceJoin is used when...FIXME

createResultProjection Method

```
createResultProjection(): (InternalRow) => InternalRow
```

createResultProjection ...FIXME

Note

createResultProjection is used when...FIXME

HashedRelation

`HashedRelation` is the [contract](#) for "relations" with values hashed by some key.

`HashedRelation` is a [KnownSizeEstimation](#).

```
package org.apache.spark.sql.execution.joins

trait HashedRelation extends KnownSizeEstimation {
    // only required methods that have no implementation
    // the others follow
    def asReadOnlyCopy(): HashedRelation
    def close(): Unit
    def get(key: InternalRow): Iterator[InternalRow]
    def getAverageProbesPerLookup: Double
    def getValue(key: InternalRow): InternalRow
    def keyIsUnique: Boolean
}
```

Note

`HashedRelation` is a `private[execution]` contract.

Table 1. HashedRelation Contract

Method	Description		
<code>asReadOnlyCopy</code>	<p>Gives a read-only copy of this <code>HashedRelation</code> to be safely used in a separate thread.</p> <p>Used exclusively when <code>BroadcastHashJoinExec</code> is requested to execute (and transform every partitions of <code>streamedPlan</code> physical operator using the broadcast variable of <code>buildPlan</code> physical operator).</p>		
<code>get</code>	<p>Gives internal rows for the given key or <code>null</code></p> <p>Used when <code>HashJoin</code> is requested to innerJoin, outerJoin, semiJoin, existenceJoin and antiJoin.</p>		
<code>getValue</code>	<p>Gives the value internal row for a given key</p> <table border="1"> <tr> <td>Note</td><td> <code>HashedRelation</code> has two variants of <code>getValue</code>, i.e. one that accepts an <code>InternalRow</code> and another a <code>Long</code>. <code>getValue</code> with an <code>InternalRow</code> does not seem to be used at all. </td></tr> </table>	Note	<code>HashedRelation</code> has two variants of <code>getValue</code> , i.e. one that accepts an <code>InternalRow</code> and another a <code>Long</code> . <code>getValue</code> with an <code>InternalRow</code> does not seem to be used at all.
Note	<code>HashedRelation</code> has two variants of <code>getValue</code> , i.e. one that accepts an <code>InternalRow</code> and another a <code>Long</code> . <code>getValue</code> with an <code>InternalRow</code> does not seem to be used at all.		
<code>getAverageProbesPerLookup</code>	Used when...FIXME		

getValue Method

```
getValue(key: Long): InternalRow
```

Note

This is `getValue` that takes a long key. There is the more generic `getValue` that takes an internal row instead.

`getValue` simply reports an `UnsupportedOperationException` (and expects concrete `HashedRelations` to provide a more meaningful implementation).

Note

`getValue` is used exclusively when `LongHashedRelation` is requested to [get the value for a given key](#).

Creating Concrete HashedRelation Instance (for Build Side of Hash-based Join) — apply Factory Method

```
apply(  
  input: Iterator[InternalRow],  
  key: Seq[Expression],  
  sizeEstimate: Int = 64,  
  taskMemoryManager: TaskMemoryManager = null): HashedRelation
```

`apply` creates a `LongHashedRelation` when the input `key` collection has a single [expression](#) of type long or `UnsafeHashedRelation` otherwise.

Note

The input `key` expressions are:

- [Build join keys](#) of `ShuffledHashJoinExec` physical operator
- [Canonicalized build-side join keys](#) of `HashedRelationBroadcastMode` (of `BroadcastHashJoinExec` physical operator)

Note

`apply` is used when:

- `ShuffledHashJoinExec` is requested to build a `HashedRelation` for given [internal rows](#)
- `HashedRelationBroadcastMode` is requested to [transform](#)

LongHashedRelation

`LongHashedRelation` is a [HashedRelation](#) that is used when `HashedRelation` is requested for a [concrete HashedRelation instance](#) when the single key is of type long.

`LongHashedRelation` is also a Java `Externalizable`, i.e. when persisted, only the identity is written in the serialization stream and it is the responsibility of the class to [save](#) and [restore](#) the contents of its instances.

`LongHashedRelation` is [created](#) when:

1. `HashedRelation` is requested for a [concrete HashedRelation](#) (and [apply](#) factory method is used)
2. `LongHashedRelation` is requested for a [read-only copy](#) (when `BroadcastHashJoinExec` is requested to [execute](#))

writeExternal Method

```
writeExternal(out: ObjectOutput): Unit
```

Note	<code>writeExternal</code> is part of Java's Externalizable Contract to...FIXME.
------	--

`writeExternal` ...FIXME

Note	<code>writeExternal</code> is used when...FIXME
------	---

readExternal Method

```
readExternal(in: ObjectInput): Unit
```

Note	<code>readExternal</code> is part of Java's Externalizable Contract to...FIXME.
------	---

`readExternal` ...FIXME

Note	<code>readExternal</code> is used when...FIXME
------	--

Creating LongHashedRelation Instance

`LongHashedRelation` takes the following when created:

- Number of fields
- LongToUnsafeRowMap

`LongHashedRelation` initializes the [internal registries and counters](#).

Creating Read-Only Copy of LongHashedRelation — `asReadOnlyCopy` Method

```
asReadOnlyCopy(): LongHashedRelation
```

Note

`asReadOnlyCopy` is part of [HashedRelation Contract](#) to...FIXME.

`asReadOnlyCopy` ...FIXME

Getting Value Row for Given Key — `getValue` Method

```
getValue(key: InternalRow): InternalRow
```

Note

`getValue` is part of [HashedRelation Contract](#) to give the value internal row for a given key.

`getValue` checks if the input `key` is null at `0` position and if so gives `null`. Otherwise, `getValue` takes the long value at position `0` and [gets the value](#).

Creating LongHashedRelation Instance — `apply` Factory Method

```
apply(
  input: Iterator[InternalRow],
  key: Seq[Expression],
  sizeEstimate: Int,
  taskMemoryManager: TaskMemoryManager): LongHashedRelation
```

`apply` ...FIXME

Note

`apply` is used exclusively when `HashedRelation` is requested for a concrete [HashedRelation](#).

UnsafeHashedRelation

`UnsafeHashedRelation` is...FIXME

get Method

```
get(key: InternalRow): Iterator[InternalRow]
```

Note	<code>get</code> is part of HashedRelation Contract to give the internal rows for the given key or <code>null</code> .
------	--

`get` ...FIXME

Getting Value Row for Given Key — `getValue` Method

```
getValue(key: InternalRow): InternalRow
```

Note	<code>getValue</code> is part of HashedRelation Contract to give the value internal row for a given key.
------	--

`getValue` ...FIXME

Creating UnsafeHashedRelation Instance — `apply` Factory Method

```
apply(  
  input: Iterator[InternalRow],  
  key: Seq[Expression],  
  sizeEstimate: Int,  
  taskMemoryManager: TaskMemoryManager): HashedRelation
```

`apply` ...FIXME

Note	<code>apply</code> is used when...FIXME
------	---

KnownSizeEstimation

`KnownSizeEstimation` is the [contract](#) that allows a class to give [SizeEstimator](#) a more accurate [size estimation](#).

`KnownSizeEstimation` defines the single `estimatedSize` method.

```
package org.apache.spark.util

trait KnownSizeEstimation {
  def estimatedSize: Long
}
```

`estimatedSize` is used when:

- `SizeEstimator` is requested to [visitSingleObject](#)
- `BroadcastExchangeExec` is requested for [relationFuture](#)
- `BroadcastHashJoinExec` is requested to [execute](#)
- `ShuffledHashJoinExec` is requested to [buildHashedRelation](#)

Note	<code>KnownSizeEstimation</code> is a <code>private[spark]</code> contract.
------	---

Note	<code>HashedRelation</code> is the only <code>KnownSizeEstimation</code> available.
------	---

SizeEstimator

`SizeEstimator` is...FIXME

estimate Method

```
estimate(obj: AnyRef): Long (1)
// Internal
estimate(obj: AnyRef, visited: IdentityHashMap[AnyRef, AnyRef]): Long
```

1. Uses `estimate` with an empty `IdentityHashMap`

`estimate` ...FIXME

Note

- | | |
|------|---|
| Note | <p><code>estimate</code> is used when:</p> <ul style="list-style-type: none"> • <code>SizeEstimator</code> is requested to <code>sampleArray</code> • FIXME |
|------|---|

sampleArray Internal Method

```
sampleArray(
  array: AnyRef,
  state: SearchState,
  rand: Random,
  drawn: OpenHashSet[Int],
  length: Int): Long
```

`sampleArray` ...FIXME

Note

Note	<code>sampleArray</code> is used when...FIXME
------	---

visitSingleObject Internal Method

```
visitSingleObject(obj: AnyRef, state: SearchState): Unit
```

`visitSingleObject` ...FIXME

Note

Note	<code>visitSingleObject</code> is used exclusively when <code>SizeEstimator</code> is requested to <code>estimate</code> .
------	--

BroadcastMode

`BroadcastMode` is the [contract](#) for...FIXME

```
package org.apache.spark.sql.catalyst.plans.physical

trait BroadcastMode {
  def canonicalized: BroadcastMode
  def transform(rows: Array[InternalRow]): Any
  def transform(rows: Iterator[InternalRow], sizeHint: Option[Long]): Any
}
```

Table 1. BroadcastMode Contract

Method	Description
<code>canonicalized</code>	Used when...FIXME
<code>transform</code>	<p>Used when:</p> <ul style="list-style-type: none"> • <code>BroadcastExchangeExec</code> is requested for relationFuture for the first time (when <code>BroadcastExchangeExec</code> is requested to prepare for execution as part of executing a physical operator) • <code>HashedRelationBroadcastMode</code> is requested to transform internal rows (and build a HashedRelation) <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>Note The <code>rows</code>-only variant does not seem to be used at all.</p> </div>

Table 2. BroadcastModes

BroadcastMode	Description
HashedRelationBroadcastMode	
IdentityBroadcastMode	

HashedRelationBroadcastMode

`HashedRelationBroadcastMode` is a [BroadcastMode](#) that `BroadcastHashJoinExec` uses for the required output distribution of child operators.

`HashedRelationBroadcastMode` takes build-side join keys (as [Catalyst expressions](#)) when created.

`HashedRelationBroadcastMode` gives a copy of itself with `keys` canonicalized when requested for a [canonicalized](#) version.

transform Method

```
transform(rows: Array[InternalRow]): HashedRelation (1)
transform(
  rows: Iterator[InternalRow],
  sizeHint: Option[Long]): HashedRelation
```

1. Uses the other `transform` with the size of `rows` as `sizeHint`

Note

`transform` is part of [BroadcastMode Contract](#) to...FIXME.

`transform` ...FIXME

IdentityBroadcastMode

IdentityBroadcastMode is a [BroadcastMode](#) that...FIXME

PartitioningUtils Helper Object

PartitioningUtils is...FIXME

validatePartitionColumn Method

```
validatePartitionColumn(  
    schema: StructType,  
    partitionColumns: Seq[String],  
    caseSensitive: Boolean): Unit
```

validatePartitionColumn ...FIXME

Note

validatePartitionColumn is used when...FIXME

parsePartitions Method

```
parsePartitions(  
    paths: Seq[Path],  
    typeInference: Boolean,  
    basePaths: Set[Path],  
    timeZoneId: String): PartitionSpec (1)  
parsePartitions(  
    paths: Seq[Path],  
    typeInference: Boolean,  
    basePaths: Set[Path],  
    timeZone: TimeZone): PartitionSpec
```

1. Uses parsePartitions with timeZoneId mapped to a TimeZone

parsePartitions ...FIXME

Note

parsePartitions is used when...FIXME

HadoopFileLinesReader

`HadoopFileLinesReader` is a Scala `Iterator` of Apache Hadoop's [org.apache.hadoop.io.Text](#).

`HadoopFileLinesReader` is [created](#) to access datasets in the following data sources:

- `SimpleTextSource`
- `LibSVMFileFormat`
- `TextInputCSVDataSource`
- `TextInputJsonDataSource`
- [TextFileFormat](#)

`HadoopFileLinesReader` uses the internal `iterator` that handles accessing files using Hadoop's FileSystem API.

Creating HadoopFileLinesReader Instance

`HadoopFileLinesReader` takes the following when created:

- [PartitionedFile](#)
- Hadoop's `Configuration`

iterator Internal Property

```
iterator: RecordReaderIterator[Text]
```

When [created](#), `HadoopFileLinesReader` creates an internal `iterator` that uses Hadoop's [org.apache.hadoop.mapreduce.lib.input.FileSplit](#) with Hadoop's [org.apache.hadoop.fs.Path](#) and `file`.

`iterator` creates Hadoop's `TaskAttemptID`, `TaskAttemptContextImpl` and `LineRecordReader`.

`iterator` initializes `LineRecordReader` and passes it on to a [RecordReaderIterator](#).

Note	<code>iterator</code> is used for <code>Iterator</code> -specific methods, i.e. <code>hasNext</code> , <code>next</code> and <code>close</code> .
------	---

CatalogUtils Helper Object

`CatalogUtils` is a Scala object with the [methods](#) to support [PreprocessTableCreation](#) post-hoc logical resolution rule (among others).

Table 1. CatalogUtils API

Name	Description
<code>maskCredentials</code>	<pre>maskCredentials(options: Map[String, String]): Map[String, String]</pre> <p>Used when:</p> <ul style="list-style-type: none"> • <code>CatalogStorageFormat</code> is requested to convert the storage specification to a LinkedHashMap • <code>CreateTempViewUsing</code> logical command is requested for the <code>argString</code>
<code>normalizeBucketSpec</code>	<pre>normalizeBucketSpec(tableName: String, tableCols: Seq[String], bucketSpec: BucketSpec, resolver: Resolver): BucketSpec</pre> <p>Used exclusively when PreprocessTableCreation post-hoc logical resolution rule is executed.</p>
<code>normalizePartCols</code>	<pre>normalizePartCols(tableName: String, tableCols: Seq[String], partCols: Seq[String], resolver: Resolver): Seq[String]</pre> <p>Used exclusively when PreprocessTableCreation post-hoc logical resolution rule is executed.</p>

normalizeColumnName Internal Method

```
normalizeColumnName(
  tableName: String,
  tableCols: Seq[String],
  colName: String,
  colType: String,
  resolver: Resolver): String
```

`normalizeColumnName` ...FIXME

Note

`normalizeColumnName` is used when `CatalogUtils` is requested to [normalizePartCols](#) and [normalizeBucketSpec](#).

ExternalCatalogUtils

ExternalCatalogUtils is...FIXME

prunePartitionsByFilter Method

```
prunePartitionsByFilter(  
    catalogTable: CatalogTable,  
    inputPartitions: Seq[CatalogTablePartition],  
    predicates: Seq[Expression],  
    defaultTimeZoneId: String): Seq[CatalogTablePartition]
```

prunePartitionsByFilter ...FIXME

Note	prunePartitionsByFilter is used when InMemoryCatalog and HiveExternalCatalog are requested to list partitions by a filter.
------	--

[InMemoryCatalog](#) and [HiveExternalCatalog](#) are requested to list partitions by a filter.

PartitioningAwareFileIndex

PartitioningAwareFileIndex is...FIXME

PartitioningAwareFileIndex uses a Hadoop Configuration for...FIXME

BufferedRowIterator

BufferedRowIterator is...FIXME

CompressionCodecs

`CompressionCodecs` is a utility object...FIXME

Table 1. Known Compression Codecs

Alias	Fully-Qualified Class Name
none	
uncompressed	
bzip2	<code>org.apache.hadoop.io.compress.BZip2Codec</code>
deflate	<code>org.apache.hadoop.io.compress.DeflateCodec</code>
gzip	<code>org.apache.hadoop.io.compress.GzipCodec</code>
lz4	<code>org.apache.hadoop.io.compress.Lz4Codec</code>
snappy	<code>org.apache.hadoop.io.compress.SnappyCodec</code>

setCodecConfiguration Method

```
setCodecConfiguration(conf: Configuration, codec: String): Unit
```

`setCodecConfiguration` sets compression-related configurations to the Hadoop Configuration per the input `codec`.

Note

The input `codec` should be a fully-qualified class name, i.e. `org.apache.hadoop.io.compress.SnappyCodec`.

If the input `codec` is defined (i.e. not `null`), `setCodecConfiguration` sets the following configuration properties.

Table 2. Compression-Related Hadoop Configuration Properties (codec defined)

Name	Value
mapreduce.output.fileoutputformat.compress	true
mapreduce.output.fileoutputformat.compress.type	BLOCK
mapreduce.output.fileoutputformat.compress.codec	The input <code>codec</code> name
mapreduce.map.output.compress	true
mapreduce.map.output.compress.codec	The input <code>codec</code> name

If the input `codec` is not defined (i.e. `null`), `setCodecConfiguration` sets the following configuration properties.

Table 3. Compression-Related Hadoop Configuration Properties (codec not defined)

Name	Value
mapreduce.output.fileoutputformat.compress	false
mapreduce.map.output.compress	false

Note

`setCodecConfiguration` is used when [CSVFileFormat](#), [JsonFileFormat](#) and [TextFileFormat](#) are requested to `preparewrite`.

SQLContext

Caution

As of Spark **2.0.0** `SQLContext` is only for backward compatibility and is a *mere wrapper* of [SparkSession](#).

In the pre-Spark 2.0's ear, **SQLContext** was the entry point for Spark SQL. Whatever you did in Spark SQL it had to start from [creating an instance of SQLContext](#).

A `SQLContext` object requires a `SparkContext`, a `CacheManager`, and a [SQLListener](#). They are all `transient` and do not participate in serializing a `SQLContext`.

You should use `SQLContext` for the following:

- [Creating Datasets](#)
- [Creating Dataset\[Long\] \(range method\)](#)
- [Creating DataFrames](#)
- [Creating DataFrames for Table](#)
- [Accessing DataFrameReader](#)
- [Accessing StreamingQueryManager](#)
- [Registering User-Defined Functions \(UDF\)](#)
- [Caching DataFrames in In-Memory Cache](#)
- [Setting Configuration Properties](#)
- [Bringing Converter Objects into Scope](#)
- [Creating External Tables](#)
- [Dropping Temporary Tables](#)
- [Listing Existing Tables](#)
- [Managing Active SQLContext for JVM](#)
- [Executing SQL Queries](#)

Creating SQLContext Instance

You can create a `SQLContext` using the following constructors:

- `SQLContext(sc: SparkContext)`
- `SQLContext.getOrCreate(sc: SparkContext)`
- `SQLContext.newSession()` allows for creating a new instance of `SQLContext` with a separate SQL configuration (through a shared `SparkContext`).

Setting Configuration Properties

You can set Spark SQL configuration properties using:

- `setConf(props: Properties): Unit`
- `setConf(key: String, value: String): Unit`

You can get the current value of a configuration property by key using:

- `getConf(key: String): String`
- `getConf(key: String, defaultValue: String): String`
- `getAllConfs: immutable.Map[String, String]`

Note	Properties that start with <code>spark.sql</code> are reserved for Spark SQL.
------	---

Creating DataFrames

emptyDataFrame

```
emptyDataFrame: DataFrame
```

`emptyDataFrame` creates an empty `DataFrame`. It calls `createDataFrame` with an empty `RDD[Row]` and an empty schema `StructType(Nil)`.

createDataFrame for RDD and Seq

```
createDataFrame[A <: Product](rdd: RDD[A]): DataFrame
createDataFrame[A <: Product](data: Seq[A]): DataFrame
```

`createDataFrame` family of methods can create a `DataFrame` from an `RDD` of Scala's Product types like case classes or tuples or `Seq` thereof.

createDataFrame for RDD of Row with Explicit Schema

```
createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame
```

This variant of `createDataFrame` creates a `DataFrame` from `RDD` of `Row` and explicit schema.

Registering User-Defined Functions (UDF)

```
udf: UDFRegistration
```

`udf` method gives you access to `UDFRegistration` to manipulate user-defined functions. Functions registered using `udf` are available for Hive queries only.

Tip

Read up on UDFs in [UDFs — User-Defined Functions](#) document.

```
// Create a DataFrame
val df = Seq("hello", "world!").zip(0 to 1).toDF("text", "id")

// Register the DataFrame as a temporary table in Hive
df.registerTempTable("texts")

scala> sql("SHOW TABLES").show
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|    texts|      true|
+-----+-----+

scala> sql("SELECT * FROM texts").show
+---+---+
| text| id|
+---+---+
| hello|  0|
|world!|  1|
+---+---+

// Just a Scala function
val my_upper: String => String = _.toUpperCase

// Register the function as UDF
spark.udf.register("my_upper", my_upper)

scala> sql("SELECT *, my_upper(text) AS MY_UPPER FROM texts").show
+---+---+-----+
| text| id|MY_UPPER|
+---+---+-----+
| hello|  0|    HELLO|
|world!|  1|    WORLD!|
+---+---+-----+
```

Caching DataFrames in In-Memory Cache

`isCached(tableName: String): Boolean`

`isCached` method asks `CacheManager` whether `tableName` table is cached in memory or not. It simply requests `CacheManager` for `CachedData` and when exists, it assumes the table is cached.

```
cacheTable(tableName: String): Unit
```

You can cache a table in memory using `cacheTable`.

Caution

Why would I want to cache a table?

```
uncacheTable(tableName: String)
clearCache(): Unit
```

`uncacheTable` and `clearCache` remove one or all in-memory cached tables.

Implicits — `SQLContext.implicitly`

The `implicitly` object is a helper class with methods to convert objects into [Datasets](#) and [DataFrames](#), and also comes with many [Encoders](#) for "primitive" types as well as the collections thereof.

Import the implicits by `import spark.implicitly._` as follows:

Note

```
val spark = new SQLContext(sc)
import spark.implicitly._
```

It holds [Encoders](#) for Scala "primitive" types like `Int`, `Double`, `String`, and their collections.

It offers support for creating `Dataset` from `RDD` of any types (for which an [encoder](#) exists in scope), or case classes or tuples, and `Seq`.

It also offers conversions from Scala's `Symbol` or `$` to `Column`.

It also offers conversions from `RDD` or `Seq` of `Product` types (e.g. case classes or tuples) to `DataFrame`. It has direct conversions from `RDD` of `Int`, `Long` and `String` to `DataFrame` with a single column name `_1`.

Note

It is not possible to call `toDF` methods on `RDD` objects of other "primitive" types except `Int`, `Long`, and `String`.

Creating Datasets

```
createDataset[T: Encoder](data: Seq[T]): Dataset[T]
createDataset[T: Encoder](data: RDD[T]): Dataset[T]
```

`createDataset` family of methods creates a [Dataset](#) from a collection of elements of type `T`, be it a regular Scala `Seq` or Spark's `RDD`.

It requires that there is an [encoder](#) in scope.

Note	Importing <code>SQLContext.implicits</code> brings many <code>encoders</code> available in scope.
------	---

Accessing DataFrameReader (read method)

```
read: DataFrameReader
```

The experimental `read` method returns a `DataFrameReader` that is used to read data from external storage systems and load it into a `DataFrame`.

Creating External Tables

```
createExternalTable(tableName: String, path: String): DataFrame
createExternalTable(tableName: String, path: String, source: String): DataFrame
createExternalTable(tableName: String, source: String, options: Map[String, String]): DataFrame
createExternalTable(tableName: String, source: String, schema: StructType, options: Map[String, String]): DataFrame
```

The experimental `createExternalTable` family of methods is used to create an external table `tableName` and return a corresponding `DataFrame`.

Caution	FIXME What is an external table?
---------	----------------------------------

It assumes **parquet** as the default data source format that you can change using `spark.sql.sources.default` setting.

Dropping Temporary Tables

```
dropTempTable(tableName: String): Unit
```

`dropTempTable` method drops a temporary table `tableName`.

Caution	FIXME What is a temporary table?
---------	----------------------------------

Creating Dataset[Long] (range method)

```
range(end: Long): Dataset[Long]
range(start: Long, end: Long): Dataset[Long]
range(start: Long, end: Long, step: Long): Dataset[Long]
range(start: Long, end: Long, step: Long, numPartitions: Int): Dataset[Long]
```

The `range` family of methods creates a `Dataset[Long]` with the sole `id` column of `LongType` for given `start`, `end`, and `step`.

Note

The three first variants use `SparkContext.defaultParallelism` for the number of partitions `numPartitions`.

```
scala> spark.range(5)
res0: org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> .show
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
+---+
```

Creating DataFrames for Table

```
table(tableName: String): DataFrame
```

`table` method creates a `tableName` table and returns a corresponding `DataFrame`.

Listing Existing Tables

```
tables(): DataFrame
tables(databaseName: String): DataFrame
```

`table` methods return a `DataFrame` that holds names of existing tables in a database.

```
scala> spark.tables.show
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|      t1|     true|
|      t2|     true|
+-----+-----+
```

The schema consists of two columns - `tableName` of `StringType` and `isTemporary` of `BooleanType`.

Note	<code>tables</code> is a result of <code>SHOW TABLES [IN databaseName]</code> .
------	---

```
tableNames(): Array[String]
tableNames(databaseName: String): Array[String]
```

`tableNames` are similar to `tables` with the only difference that they return `Array[String]` which is a collection of table names.

Accessing StreamingQueryManager

```
streams: StreamingQueryManager
```

The `streams` method returns a [StreamingQueryManager](#) that is used to...TK

Caution	FIXME
---------	-------

Managing Active SQLContext for JVM

```
SQLContext.getOrCreate(sparkContext: SparkContext): SQLContext
```

`SQLContext.getOrCreate` method returns an active `SQLContext` object for the JVM or creates a new one using a given `sparkContext` .

Note	It is a factory-like method that works on <code>SQLContext</code> class.
------	--

Interestingly, there are two helper methods to set and clear the active `SQLContext` object - `setActive` and `clearActive` respectively.

```
setActive(spark: SQLContext): Unit
clearActive(): Unit
```

Executing SQL Queries

```
sql(sqlText: String): DataFrame
```

`sql` executes the `sqlText` SQL query.

Note	It supports Hive statements through HiveContext .
------	---

```

scala> sql("set spark.sql.hive.version").show(false)
16/04/10 15:19:36 INFO HiveSqlParser: Parsing command: set spark.sql.hive.version
+-----+-----+
|key |value|
+-----+-----+
|spark.sql.hive.version|1.2.1|
+-----+-----+


scala> sql("describe database extended default").show(false)
16/04/10 15:21:14 INFO HiveSqlParser: Parsing command: describe database extended defa
ult
+-----+-----+
|database_description_item|database_description_value|
+-----+-----+
|Database Name |default |
|Description |Default Hive database |
|Location |file:/user/hive/warehouse |
|Properties | |
+-----+-----+


// Create temporary table
scala> spark.range(10).registerTempTable("t")
16/04/14 23:34:31 INFO HiveSqlParser: Parsing command: t

scala> sql("CREATE temporary table t2 USING PARQUET OPTIONS (PATH 'hello') AS SELECT * 
   FROM t")
16/04/14 23:34:38 INFO HiveSqlParser: Parsing command: CREATE temporary table t2 USING
 PARQUET OPTIONS (PATH 'hello') AS SELECT * FROM t

scala> spark.tables.show
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|      t|     true|
|     t2|     true|
+-----+-----+

```

`sql` parses `sqlText` using a dialect that can be set up using `spark.sql.dialect` setting.

Note

`sql` is imported in spark-shell so you can execute Hive statements without `spark` prefix.

```

scala> :imports
1) import spark.implicits._ (52 terms, 31 are implicit)
2) import spark.sql          (1 terms)

```

Tip

You may also use `spark-sql shell script` to interact with Hive.

Internally, it uses `SessionState.sqlParser.parsePlan(sql)` method to create a `LogicalPlan`.

Caution

FIXME Review

```
scala> sql("show tables").show(false)
16/04/09 13:05:32 INFO HiveSqlParser: Parsing command: show tables
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|data    |false     |
+-----+-----+
```

Enable `INFO` logging level for the loggers that correspond to the `AbstractSqlParser` to see what happens inside `sql`.

Add the following line to `conf/log4j.properties`:

Tip

```
log4j.logger.org.apache.spark.sql.hive.execution.HiveSqlParser=INFO
```

Refer to [Logging](#).

Creating New Session

```
newSession(): SQLContext
```

You can use `newSession` method to create a new session without a cost of instantiating a new `SQLContext` from scratch.

`newSession` returns a new `SQLContext` that shares `sparkContext`, `CacheManager`, `SQLListener`, and [ExternalCatalog](#).

Caution

FIXME Why would I need that?