The Internals of Spark SQL

None

Jacek Laskowski

Table of contents

1. The Internals of Spark SQL (Apache Spark 3.5.3)

Welcome to **The Internals of Spark SQL** online book! 🤙

I'm Jacek Laskowski, a Freelance Data(bricks) Engineer specializing in Apache Spark (incl. Spark SQL and Spark Structured Streaming), Delta Lake, Databricks, and Apache Kafka (incl. Kafka Streams) with brief forays into a wider data engineering space (e.g., Trino, Dask and dbt, mostly during Warsaw Data Engineering meetups).

I'm very excited to have you here and hope you will enjoy exploring the internals of Spark SQL as much as I have.

> **Flannery O'Connor**
>
> I write to discover what I know.

> **"The Internals Of" series**
>
> I'm also writing other online books in the "The Internals Of" series. Please visit "The Internals Of" Online Books home page.

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

Now, let's take a deep dive into Spark SQL 🔥

Last update: 2024-11-30

2. Features

2.1 Features

The following are the features of Spark SQL that help place it in the top of the modern distributed SQL query processing engines:

- Adaptive Query Execution
- Bloom Filter Join
- Catalog Plugin API
- Columnar Execution
- Connector API
- Default Columns
- Direct Queries on Files
- Dynamic Partition Pruning
- File-Based Data Scanning
- Hints
- Metadata Columns
- Named Function Arguments
- Spark Connect
- Table-Valued Functions
- Time Travel
- Variable Substitution
- Vectorized Parquet Decoding (Reader)
- Whole-Stage Code Generation
- *others* (listed in the menu on the left)

2.2 Aggregate Queries

2.2.1 Aggregate Queries

Aggregate Queries (*Aggregates*) are structured queries with Aggregate logical operator.

Aggregate Queries calculate single value for a set of rows.

Aggregate Queries can be broken down to the following sections:

1. **Grouping** (using `GROUP BY` clause in SQL or Dataset.groupBy operator) that arranges rows into groups (possibly guarded by HAVING SQL clause)
2. **Aggregation** (using Aggregate Functions) to apply to a set of rows and calculate single values per groups

Whole-Stage Code Generation

Whole-Stage Code Generation is supported by AggregateCodegenSupport physical operators only with supportCodegen flag enabled.

Adaptive Query Execution

Adaptive Query Execution uses ReplaceHashWithSortAgg physical optimization among the queryStagePreparationRules.

Configuration Properties

Aggregate Queries can be fine-tuned with the following configuration properties:

- spark.sql.execution.replaceHashWithSortAgg
- spark.sql.retainGroupColumns
- *others*

High-Level Operators

`Aggregate` is a logical representation of the high-level operators in SQL or Dataset API.

SQL

`Aggregate` represents the following SQL clauses:

- GROUP BY (incl. `GROUPING SETS`, `WITH CUBE`, `WITH ROLLUP`)
- visitCommonSelectQueryClausePlan

DATASET

`Aggregate` represents the following high-level operators in Dataset API:

- KeyValueGroupedDataset.agg
- RelationalGroupedDataset.agg
- RelationalGroupedDataset.avg
- RelationalGroupedDataset.count
- RelationalGroupedDataset.max
- RelationalGroupedDataset.mean
- RelationalGroupedDataset.min
- RelationalGroupedDataset.sum

Group Types

`GroupType` indicates the kind of an aggregation.

CUBE

GROUPBY

PIVOT

ROLLUP

UnsupportedOperationChecker

`UnsupportedOperationChecker` is responsible for asserting correctness of aggregation queries (among others).

> **FIXME List unsupported features**

Basic Aggregation

Basic Aggregation calculates aggregates over a group of rows using aggregate operators (possibly with aggregate functions).

Multi-Dimensional Aggregation

Multi-Dimensional Aggregate Operators are variants of groupBy operator to create queries for subtotals, grand totals and superset of subtotals in one go.

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.

Beside Dataset.cube and Dataset.rollup operators, Spark SQL supports GROUPING SETS clause in SQL mode only.

> **SPARK-6356**
>
> Support for multi-dimensional aggregate operators was added in [SPARK-6356] Support the ROLLUP/CUBE/GROUPING SETS/grouping() in SQLContext.

Aggregate Operators

AGG

Aggregates over (*applies an aggregate function on*) a subset of or the entire `Dataset` (i.e., considering the entire data set as one group)

Creates a RelationalGroupedDataset

> **Note**
>
> `Dataset.agg` is simply a shortcut for `Dataset.groupBy().agg`.

CUBE

```
cube(
  cols: Column*): RelationalGroupedDataset
cube(
  col1: String,
  cols: String*): RelationalGroupedDataset
```

```
GROUP BY expressions WITH CUBE
GROUP BY CUBE(expressions)
```

`cube` multi-dimensional aggregate operator returns a RelationalGroupedDataset to calculate subtotals and a grand total for every permutation of the columns specified.

`cube` 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.

cube vs rollup

`cube` is more than rollup operator, i.e. `cube` does `rollup` with aggregation over all the missing combinations given the columns.

GROUPBY

Groups the rows in a `Dataset` by columns (as Column expressions or names).

Creates a RelationalGroupedDataset

Used for **untyped aggregates** using `DataFrame`s. Grouping is described using column expressions or column names.

GROUPBYKEY

Groups records (of type `T`) by the input `func` and creates a KeyValueGroupedDataset to apply aggregation to.

Used for **typed aggregates** using `Dataset`s with records grouped by a key-defining discriminator function

```
import org.apache.spark.sql.expressions.scalalang._
val q = dataset
  .groupByKey(_.productId).
  .agg(typed.sum[Token](_.score))
  .toDF("productId", "sum")
  .orderBy('productId)
```

```
spark
  .readStream
  .format("rate")
  .load
  .as[(Timestamp, Long)]
  .groupByKey { case (ts, v) => v % 2 }
  .agg()
  .writeStream
  .format("console")
  .trigger(Trigger.ProcessingTime(5.seconds))
  .outputMode("complete")
  .start
```

GROUPING SETS

```
GROUP BY (expressions) GROUPING SETS (expressions)
GROUP BY GROUPING SETS (expressions)
```

Note

SQL's `GROUPING SETS` is the most general aggregate "operator" and can generate the same dataset as using a simple groupBy, cube and rollup 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|
+----+-----+------+
```

`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|
+-------+----+------+
```

`GROUPING SETS` clause is parsed in withAggregation parsing handler (in `AstBuilder`) and becomes a GroupingSets logical operator internally.

ROLLUP

```
rollup(
  cols: Column*): RelationalGroupedDataset
rollup(
```

```
  col1: String,
  cols: String*): RelationalGroupedDataset
```

```
GROUP BY expressions WITH ROLLUP
GROUP BY ROLLUP(expressions)
```

`rollup` gives a RelationalGroupedDataset to calculate subtotals and a grand total over (ordered) combination of groups.

`rollup` 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 https://www.postgresql.org/docs/current/static/queries-table-expressions.html#QUERIES-GROUPING-SETS[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`).

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.

```
// 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 and then creates a RelationalGroupedDataset (with `RollupType` group type).

Tip

Read up on `rollup` in Deeper into Postgres 9.5 - New Group By Options for Aggregation.

Catalyst DSL

Catalyst DSL defines groupBy operator to create aggregation queries.

Aggregate Query Execution

LOGICAL ANALYSIS

The following logical analysis rules handle `Aggregate` logical operator:

- CleanupAliases
- `ExtractGenerator`
- `ExtractWindowExpressions`
- `GlobalAggregates`
- ResolveAliases
- ResolveGroupingAnalytics
- ResolveOrdinalInOrderByAndGroupBy
- `ResolvePivot`

LOGICAL OPTIMIZATIONS

The following logical optimizations handle `Aggregate` logical operator:

- `DecorrelateInnerQuery`
- `InjectRuntimeFilter`
- `MergeScalarSubqueries`
- OptimizeMetadataOnlyQuery
- `PullOutGroupingExpressions`
- PullupCorrelatedPredicates
- `ReplaceDistinctWithAggregate`
- `ReplaceDeduplicateWithAggregate`
- `RewriteAsOfJoin`
- RewriteCorrelatedScalarSubquery
- `RewriteDistinctAggregates`
- RewriteExceptAll
- `RewriteIntersectAll`
- V2ScanRelationPushDown

Cost-Based Optimization

`Aggregate` operators are handled by BasicStatsPlanVisitor for visitDistinct and visitAggregate

PushDownPredicate

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 `count`#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 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]
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]
```

PHYSICAL OPTIMIZATIONS

The following physical optimizations use `Aggregate` logical operator:

- PlanAdaptiveDynamicPruningFilters
- PlanDynamicPruningFilters
- `RowLevelOperationRuntimeGroupFiltering`

ReplaceHashWithSortAgg

ReplaceHashWithSortAgg physical optimization can replace HashAggregateExec and ObjectHashAggregateExec physical operators with SortAggregateExec when executed with spark.sql.execution.replaceHashWithSortAgg configuration property and *some sorting requirements* are met.

QUERY PLANNING

Aggregation execution planning strategy is used to plan Aggregate logical operators for execution as one of the available BaseAggregateExec physical operators:

- HashAggregateExec
- ObjectHashAggregateExec
- SortAggregateExec

Demo

Demo: Mult-Dimensional Aggregations

2.2.2 AggUtils Utility

`AggUtils` is an utility for Aggregation execution planning strategy.

planAggregateWithoutDistinct

```
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**
>
> `requiredChildDistributionExpressions` 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**
>
> `requiredChildDistributionExpressions` for the parent aggregate physical operator for final aggregation "stage" are the Attributes of the `groupingExpressions`.

planAggregateWithOneDistinct

```
planAggregateWithOneDistinct(
  groupingExpressions: Seq[NamedExpression],
  functionsWithDistinct: Seq[AggregateExpression],
  functionsWithoutDistinct: Seq[AggregateExpression],
  resultExpressions: Seq[NamedExpression],
  child: SparkPlan): Seq[SparkPlan]
```

`planAggregateWithOneDistinct` ...FIXME

Creating Physical Operator for Aggregation

```
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 one of the following physical operators based on the given AggregateExpressions (in the following order):

1. HashAggregateExec when all the aggBufferAttributes (of the AggregateFunctions of the given AggregateExpressions) are supported
2. ObjectHashAggregateExec when the following all hold:
   - spark.sql.execution.useObjectHashAggregateExec configuration property is enabled
   - Aggregate expression supported
3. SortAggregateExec

`createAggregate` is used when:

- `AggUtils` is used to createStreamingAggregate, planAggregateWithoutDistinct, planAggregateWithOneDistinct

Planning Execution of Streaming Aggregation

```
planStreamingAggregation(
  groupingExpressions: Seq[NamedExpression],
  functionsWithoutDistinct: Seq[AggregateExpression],
  resultExpressions: Seq[NamedExpression],
  stateFormatVersion: Int,
  child: SparkPlan): Seq[SparkPlan]
```

`planStreamingAggregation` ...FIXME

`planStreamingAggregation` is used when:

- `StatefulAggregationStrategy` (Spark Structured Streaming) execution planning strategy is requested to plan a logical plan of a streaming aggregation (a streaming query with Aggregate operator)

Creating Streaming Aggregate Physical Operator

```
createStreamingAggregate(
  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
```

`createStreamingAggregate` creates an aggregate physical operator (with `isStreaming` flag enabled).

> **Note**
>
> `createStreamingAggregate` is exactly createAggregate with `isStreaming` flag enabled.

`createStreamingAggregate` is used when:

- `AggUtils` is requested to plan a regular and session-windowed streaming aggregation

2.2.3 AggregationIterator

`AggregationIterator` is an abstraction of aggregation iterators (of UnsafeRows) that are used by aggregate physical operators to process rows in a partition.

```
abstract class AggregationIterator(...)
extends Iterator[UnsafeRow]
```

From scala.collection.Iterator:

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.

Implementations

- ObjectAggregationIterator
- SortBasedAggregationIterator
- TungstenAggregationIterator

Creating Instance

`AggregationIterator` takes the following to be created:

- Partition ID
- Grouping NamedExpressions
- Input Attributes
- AggregateExpressions
- Aggregate Attributes
- Initial input buffer offset
- Result NamedExpressions
- Function to create a new `MutableProjection` given expressions and attributes (`(Seq[Expression], Seq[Attribute]) => MutableProjection`)

Abstract Class

`AggregationIterator` is an abstract class and cannot be created directly. It is created indirectly for the concrete AggregationIterators.

AggregateModes

When created, `AggregationIterator` makes sure that there are at most 2 distinct `AggregateMode` s of the AggregateExpressions.

The `AggregateMode` s have to be a subset of the following mode pairs:

- `Partial` and `PartialMerge`
- `Final` and `Complete`

Process Row Function

```
processRow: (InternalRow, InternalRow) => Unit
```

`AggregationIterator` generates a `processRow` function when created.

`processRow` is a procedure

`processRow` is a procedure that takes two InternalRows and produces no output (returns `Unit`).

`processRow` is similar to the following definition:

```
def processRow(currentBuffer: InternalRow, row: InternalRow): Unit = {
  ...
}
```

`AggregationIterator` uses the aggregateExpressions, the aggregateFunctions and the inputAttributes to generate the processRow procedure.

`processRow` is used when:

- `MergingSessionsIterator` is requested to `processCurrentSortedGroup`
- `ObjectAggregationIterator` is requested to process input rows
- `SortBasedAggregationIterator` is requested to processCurrentSortedGroup
- `TungstenAggregationIterator` is requested to process input rows

AggregateFunctions

```
aggregateFunctions: Array[AggregateFunction]
```

When created, `AggregationIterator` initializes AggregateFunctions in the aggregateExpressions (with initialInputBufferOffset).

initializeAggregateFunctions

```
initializeAggregateFunctions(
  expressions: Seq[AggregateExpression],
  startingInputBufferOffset: Int): Array[AggregateFunction]
```

`initializeAggregateFunctions` ...FIXME

`initializeAggregateFunctions` is used when:

- `AggregationIterator` is requested for the aggregateFunctions
- `ObjectAggregationIterator` is requested for the mergeAggregationBuffers
- `TungstenAggregationIterator` is requested to switchToSortBasedAggregation

Generate Output Function

```
generateOutput: (UnsafeRow, InternalRow) => UnsafeRow
```

`AggregationIterator` creates a ResultProjection function when created.

`generateOutput` is used by the aggregate iterators when they are requested for the next element (aggregate result) and generate an output for empty grouping with no input.

Aggregate Iterators	Operations
`ObjectAggregationIterator`	• next element • outputForEmptyGroupingKeyWithoutInput
`SortBasedAggregationIterator`	• next element • outputForEmptyGroupingKeyWithoutInput
`TungstenAggregationIterator`	• next element • outputForEmptyGroupingKeyWithoutInput

GENERATING RESULT PROJECTION

```
generateResultProjection(): (UnsafeRow, InternalRow) => UnsafeRow
```

TungstenAggregationIterator

TungstenAggregationIterator overrides generateResultProjection for partial aggregation (non-`Final` and non-`Complete` aggregate modes).

`generateResultProjection` branches off based on the aggregate modes of the aggregates:

1. Final and Complete
2. Partial and PartialMerge
3. No modes

Main Differences between Aggregate Modes

Final and Complete	Partial and PartialMerge
Focus on DeclarativeAggregates to execute the evaluateExpressions (while the allImperativeAggregateFunctions simply eval)	Focus on TypedImperativeAggregates so they can serializeAggregateBufferInPlace
An UnsafeProjection binds the resultExpressions to the following: 1. groupingAttributes 2. the aggregateAttributes	An UnsafeProjection binds the groupingAttributes and bufferAttributes to the following (repeated twice rightly): 1. the groupingAttributes 2. the bufferAttributes
Uses an UnsafeProjection to generate an UnsafeRow for the following: 1. the current grouping key 2. the aggregate results	Uses an UnsafeProjection to generate an UnsafeRow for the following: 1. the current grouping key 2. the current buffer

Final and Complete

For Final or Complete modes, `generateResultProjection` does the following:

1. Collects expressions to evaluate the final values of the DeclarativeAggregates and `NoOp`s for the AggregateFunctions among the aggregateFunctions. `generateResultProjection` preserves the order of the evaluate expressions and `NoOp`s (so the `i`th aggregate function uses the `i`th evaluation expressions)
2. Executes the newMutableProjection with the evaluation expressions and the aggBufferAttributes of the aggregateFunctions to create a MutableProjection
3. Requests the `MutableProjection` to store the aggregate results (of all the DeclarativeAggregates) in a `SpecificInternalRow`
4. Creates an UnsafeProjection for the resultExpressions and the groupingAttributes with the aggregateAttributes (for the input schema)
5. Initializes the `UnsafeProjection` with the partIndex

In the end, `generateResultProjection` creates a result projection function that does the following:

1. Generates results for all expression-based aggregate functions (using the `MutableProjection` with the given `currentBuffer`)
2. Generates results for all imperative aggregate functions
3. Uses the `UnsafeProjection` to generate an UnsafeRow with the aggregate results for the current grouping key and the aggregate results

Partial and PartialMerge

For Partial or PartialMerge modes, `generateResultProjection` does the following:

1. Creates an UnsafeProjection for the groupingAttributes with the aggBufferAttributes of the aggregateFunctions
2. Initializes the `UnsafeProjection` with the partIndex
3. Collects the TypedImperativeAggregates from the aggregateFunctions (as they store a generic object in an aggregation buffer, and require calling serialization before shuffling)

In the end, `generateResultProjection` creates a result projection function that does the following:

1. Requests the TypedImperativeAggregates (from the aggregateFunctions) to serializeAggregateBufferInPlace with the given `currentBuffer`
2. Uses the `UnsafeProjection` to generate an UnsafeRow with the current grouping key and buffer

No Modes

For no aggregate modes, `generateResultProjection` ...FIXME

Initializing Aggregation Buffer

```
initializeBuffer(
  buffer: InternalRow): Unit
```

`initializeBuffer` requests the expressionAggInitialProjection to store an execution result of an empty row in the given InternalRow (`buffer`).

`initializeBuffer` requests all the ImperativeAggregate functions to initialize with the `buffer` internal row.

`initializeBuffer` is used when:

- `MergingSessionsIterator` is requested to `newBuffer`, `initialize`, `next`, `outputForEmptyGroupingKeyWithoutInput`
- `SortBasedAggregationIterator` is requested to newBuffer, initialize, next and outputForEmptyGroupingKeyWithoutInput

Generating Process Row Function

```
generateProcessRow(
  expressions: Seq[AggregateExpression],
  functions: Seq[AggregateFunction],
  inputAttributes: Seq[Attribute]): (InternalRow, InternalRow) => Unit
```

`generateProcessRow` creates a mutable `JoinedRow` (of two InternalRows).

`generateProcessRow` branches off based on the given AggregateExpressions, specified or not.

Where AggregateExpressions come from

Caller	AggregateExpressions
AggregationIterator	aggregateExpressions
ObjectAggregationIterator	aggregateExpressions
TungstenAggregationIterator	aggregateExpressions

`functions` Argument

`generateProcessRow` works differently based on the type of the given AggregateFunctions:

- DeclarativeAggregate
- AggregateFunction
- ImperativeAggregate

`generateProcessRow` is used when:

- `AggregationIterator` is requested for the process row function
- `ObjectAggregationIterator` is requested for the mergeAggregationBuffers function
- `TungstenAggregationIterator` is requested to switch to sort-based aggregation

AGGREGATE EXPRESSIONS SPECIFIED

Merge Expressions

With AggregateExpressions specified, `generateProcessRow` determines so-called "merge expressions" (`mergeExpressions`) as follows:

- For DeclarativeAggregate functions, the merge expressions are choosen based on the AggregateMode of the corresponding AggregateExpression

AggregateMode	Merge Expressions
`Partial` or `Complete`	Update Expressions of a `DeclarativeAggregate`
`PartialMerge` or `Final`	Merge Expressions of a `DeclarativeAggregate`

- For AggregateFunction functions, there are as many `NoOp` merge expressions (that do nothing and do not change a value) as there are aggBufferAttributes in a `AggregateFunction`

Initialize Predicates

`generateProcessRow` finds AggregateExpressions with filters specified.

When in `Partial` or `Complete` aggregate modes, `generateProcessRow` ...FIXME

Update Functions

`generateProcessRow` determines so-called "update functions" (`updateFunctions`) among ImperativeAggregate functions (in the given AggregateFunctions) to be as follows:

- FIXME

Update Projection

`generateProcessRow` uses the newMutableProjection generator function to create a MutableProjection based on the `mergeExpressions` and the aggBufferAttributes of the given AggregateFunctions with the given `inputAttributes`.

Process Row Function

In the end, `generateProcessRow` creates a procedure that accepts two InternalRows (`currentBuffer` and `row`) that does the following:

1. Processes all expression-based aggregate functions (using `updateProjection`). `generateProcessRow` requests the MutableProjection to store the output in the `currentBuffer`. The output is created based on the `currentBuffer` and the `row`.
2. Processes all imperative aggregate functions. `generateProcessRow` requests every "update function" (in `updateFunctions`) to execute with the given `currentBuffer` and the `row`.

NO AGGREGATE EXPRESSIONS

With no AggregateExpressions (`expressions`), `generateProcessRow` creates a function that does nothing ("swallows" the input).

2.2.4 KVSorterIterator

`KVSorterIterator` is...FIXME

2.2.5 ObjectAggregationIterator

`ObjectAggregationIterator` is an AggregationIterator for ObjectHashAggregateExec physical operator.

Creating Instance

`ObjectAggregationIterator` takes the following to be created:

- Partition ID
- Output Attributes (unused)
- Grouping NamedExpressions
- AggregateExpressions
- Aggregate Attributes
- Initial input buffer offset
- Result NamedExpressions
- Function to create a new `MutableProjection` given expressions and attributes (`(Seq[Expression], Seq[Attribute]) => MutableProjection`)
- Original Input Attributes
- Input InternalRows
- spark.sql.objectHashAggregate.sortBased.fallbackThreshold
- numOutputRows metric
- spillSize metric
- numTasksFallBacked metric

While being created, `ObjectAggregationIterator` starts processing input rows.

`ObjectAggregationIterator` is created when:

- `ObjectHashAggregateExec` physical operator is requested to doExecute

outputForEmptyGroupingKeyWithoutInput

```
outputForEmptyGroupingKeyWithoutInput(): UnsafeRow
```

`outputForEmptyGroupingKeyWithoutInput` ...FIXME

`outputForEmptyGroupingKeyWithoutInput` is used when:

- `ObjectHashAggregateExec` physical operator is executed (with no input rows and no groupingExpressions)

Processing Input Rows

```
processInputs(): Unit
```

`processInputs` creates an ObjectAggregationMap.

For no groupingExpressions, `processInputs` uses the groupingProjection to generate a grouping key (for `null` row) and finds the aggregation buffer that is used to process all input rows (of a partition).

Otherwise, `processInputs` uses the sortBased flag to determine whether to use the `ObjectAggregationMap` or switch to a `SortBasedAggregator`.

`processInputs` uses the groupingProjection to generate a grouping key for an input row and finds the aggregation buffer that is used to process the row (of a partition). `processInputs` continues processing input rows until there are no more rows available or the size of the `ObjectAggregationMap` reaches spark.sql.objectHashAggregate.sortBased.fallbackThreshold.

When the size of the `ObjectAggregationMap` reaches spark.sql.objectHashAggregate.sortBased.fallbackThreshold and there are still input rows in the partition, `processInputs` prints out the following INFO message to the logs, turns the sortBased flag on and increments the numTasksFallBacked metric.

```
Aggregation hash map size [size] reaches threshold capacity ([fallbackCountThreshold] entries),
spilling and falling back to sort based aggregation.
You may change the threshold by adjusting the option spark.sql.objectHashAggregate.sortBased.fallbackThreshold
```

For sort-based aggregation (the sortBased flag is enabled), `processInputs` requests the `ObjectAggregationMap` to dumpToExternalSorter and create a `KVSorterIterator`. `processInputs` creates a `SortBasedAggregator`, uses the groupingProjection to generate a grouping key for every input row and adds them to the `SortBasedAggregator`.

In the end, `processInputs` creates the aggBufferIterator (from the `ObjectAggregationMap` or `SortBasedAggregator` based on the sortBased flag).

`processInputs` is used when:

- `ObjectAggregationIterator` is created

Logging

Enable `ALL` logging level for `org.apache.spark.sql.execution.aggregate.ObjectAggregationIterator` logger to see what happens inside.

Add the following line to `conf/log4j2.properties`:

```
log4j.logger.org.apache.spark.sql.execution.aggregate.ObjectAggregationIterator=ALL
```

Refer to Logging.

2.2.6 ObjectAggregationMap

`ObjectAggregationMap` is an in-memory map to store aggregation buffer for hash-based aggregation (using ObjectAggregationIterator).

2.2.7 PhysicalAggregation Scala Extractor

`PhysicalAggregation` is a Scala extractor to destructure an Aggregate logical operator into a four-element tuple (`ReturnType`) with the following elements:

1. NamedExpressions of the grouping keys
2. AggregateExpressions
3. NamedExpressions of the result
4. Child logical operator

ReturnType

```
(Seq[NamedExpression], Seq[AggregateExpression], Seq[NamedExpression], LogicalPlan)
```

Scala Extractor Object

Learn more in the Scala extractor objects.

Destructuring Aggregate Logical Operator

```
type ReturnType =
  (Seq[NamedExpression],      // Grouping Keys
   Seq[AggregateExpression],  // Aggregate Functions
   Seq[NamedExpression],      // Result
   LogicalPlan)               // Child

unapply(
  a: Any): Option[ReturnType]
```

`unapply` destructures an Aggregate logical operator into a four-element ReturnType tuple.

`unapply` creates a EquivalentExpressions (to eliminate duplicate aggregate expressions and avoid evaluating them multiple times).

`unapply` collects AggregateExpressions in the resultExpressions of the given Aggregate logical operator.

Some Other Magic

`unapply` does *some other magic* but it does not look interesting, but the main idea should already be explained 😉

`unapply` is used when:

- `StatefulAggregationStrategy` (Spark Structured Streaming) execution planning strategy is executed
- Aggregation execution planning strategy is executed

2.2.8 SortBasedAggregationIterator

`SortBasedAggregationIterator` is an AggregationIterator that is used by SortAggregateExec physical operator to process rows in a partition.

Creating Instance

`SortBasedAggregationIterator` takes the following to be created:

- Partition ID
- Grouping NamedExpressions
- Value Attributes
- Input Iterator (InternalRows)
- AggregateExpressions
- Aggregate Attributes
- Initial input buffer offset
- Result NamedExpressions
- Function to create a new `MutableProjection` given expressions and attributes (`(Seq[Expression], Seq[Attribute]) => MutableProjection`)
- number of output rows metric

`SortBasedAggregationIterator` initializes immediately.

`SortBasedAggregationIterator` is created when:

- `SortAggregateExec` physical operator is requested to doExecute

INITIALIZATION

```
initialize(): Unit
```

Procedure

`initialize` is a procedure (returns `Unit`) so *what happens inside stays inside* (paraphrasing the former advertising slogan of Las Vegas, Nevada).

`initialize` ...FIXME

Performance Metrics

`SortBasedAggregationIterator` is given the performance metrics of the owning SortAggregateExec aggregate physical operator when created.

The metrics are displayed as part of SortAggregateExec aggregate physical operator (e.g. in web UI in Details for Query).