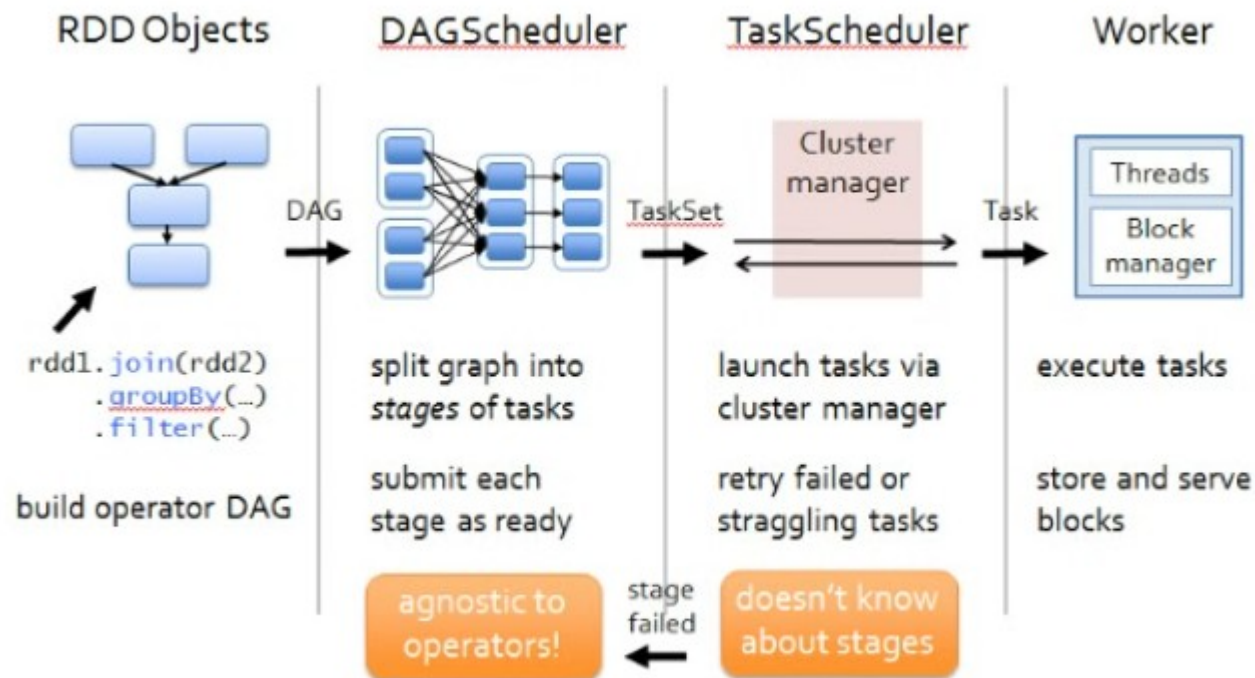


第四课：SparkSQL 原理和实践

【声明】 本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

<http://edu.dataguru.cn>



■ Spark 的运行架构

■ 调度器

- TaskScheduler
- YarnClusterScheduler
- YarnClientClusterScheduler

- hive 原理和架构
- hive 演示

- shark 原理和架构
- shark 演示

- SparkSQL 原理和架构
- SparkSQL 演示



- **hive 原理和架构**
- **hive 演示**

- **shark 原理和架构**
- **shark 演示**

- **SparkSQL 原理和架构**
- **SparkSQL 演示**



什么是 hive

- 由 facebook 开源，最初用于解决海量结构化的日志数据统计问题；
 - ETL(Extraction-Transformation-Loading) 工具
- 构建于 hadoop 的 hdfs 和 mapred 之上，用于管理和查询结构化 / 非结构化数据的数据仓库
- 设计目的是让 SQL 技能良好，但 Java 技能较弱的分析师可以查询海量数据
 - 使用 HQL 作为查询接口
 - 使用 HDFS 作为底层存储
 - 使用 MapRed 作为执行层
- 2008 年 facebook 把 hive 项目贡献给 Apache



大数据的挑战

- 海量数据时代的到来
 - IDC 数据表明,全球企业数据正以 55% 的速度逐年增长,IDC 预计,到 2020 年,全球数字信息总量将增长 44 倍。以某网络视频公司为例:每天新增数据量高达 500G。
- 非结构化数据的爆炸式增长
 - 有超过 80% 的数据都是非结构化的,如网站访问日志、移动互联网数据和聊天交流工具记录等。
- 存储和查询分析需要
 - 愈加激烈的竞争要求对客户进行更加深入细致的分析。
- 传统技术无法胜任大数据的存储、管理、分析和挖掘
 - 传统的关系型数据库以及 BI 分析工具通常只能处理 GB 级别的结构化数据

Hadoop生态@爱奇艺

- 处理数据量: ~20PB
- 日处理Job数: 100000+
- 服务项目组: 搜索、广告、推荐、日志分析、BI等。

数据业务规模 (截至2014.4)



- 每日新增数据量: 5T(压缩后)
- ETL任务: 1800
- 每日Hadoop Job 数: 20000+
- 报表数量: 985
- 支撑RD和分析师: 460+

大数据的挑战

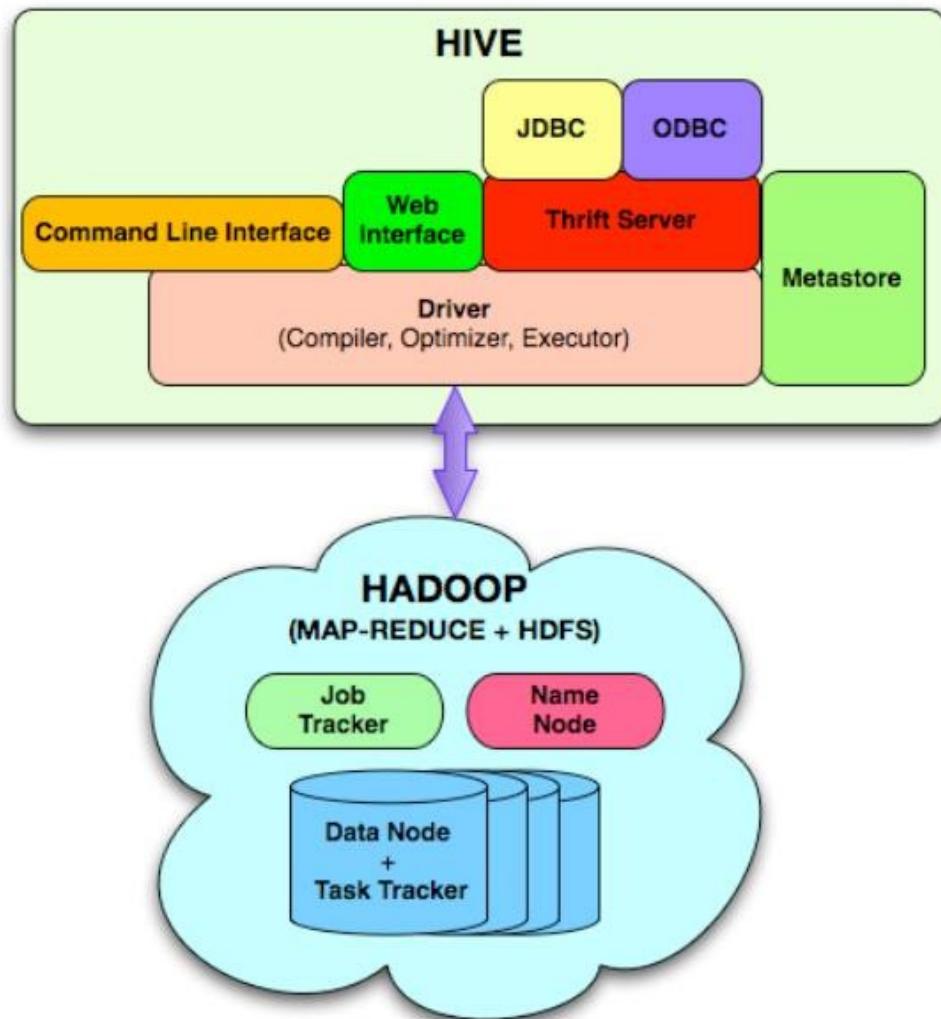
- 虽然 Hadoop 的 hdfs 和 mapred 已经能够很好的解决大数据的存储和分析问题，但是对于传统的数据分析人员来说，他们还面临着以下挑战：
 - 理解 mapred 计算模型
 - 自行开发代码实现业务逻辑
- Hive 的出现，完美的解决了传统数据分析人员所面临的问题。
 - Hive 使用类 SQL 查询语法，最大限度的实现了和 SQL 标准的兼容。
 - JDBC 接口 / ODBC 接口也使开发人员更易开发应用
- 为超大数据集设计的计算 / 扩展能力
 - MR 作为计算引擎，HDFS 作为存储系统
- 统一的元数据管理
 - 可与 Pig、Presto 等共享



- Hive 的 HQL 表达的能力有限
 - 有些复杂运算用 HQL 不易表达
- Hive 效率较低
 - Hive 自动生成 MapReduce 作业, 通常不够智能;
 - HQL 调优困难, 粒度较粗
 - 可控性差



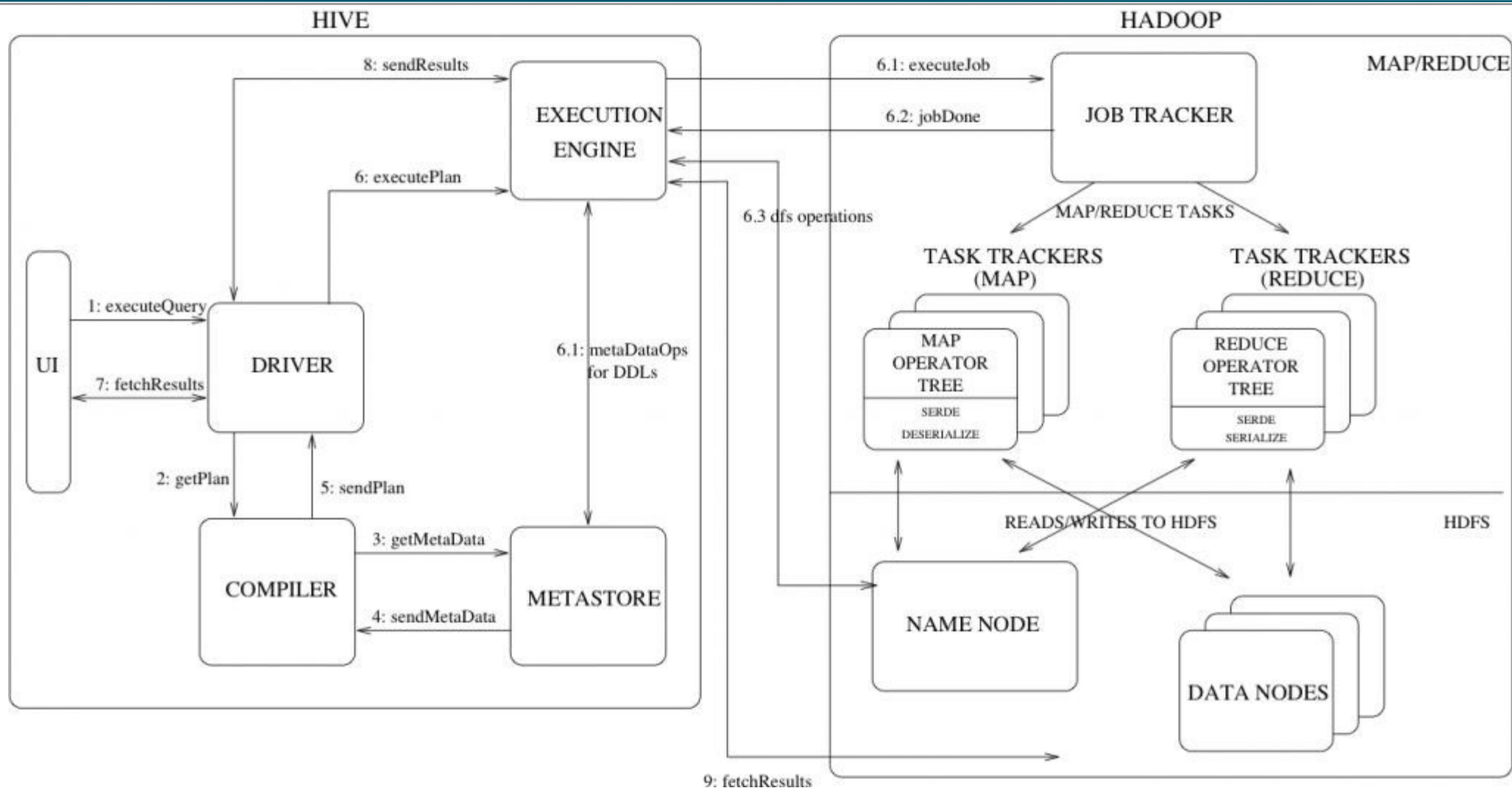
hive 的运行架构



hive 系统架构

- 元数据存储 (Metastore)
- 驱动 (Driver)
 - 编译器
 - 优化器
 - 执行器
- 接口
 - CLI
 - HWI
 - ThriftServer
- Hadoop
 - 用 MapReduce 进行计算
 - 用 HDFS 进行存储

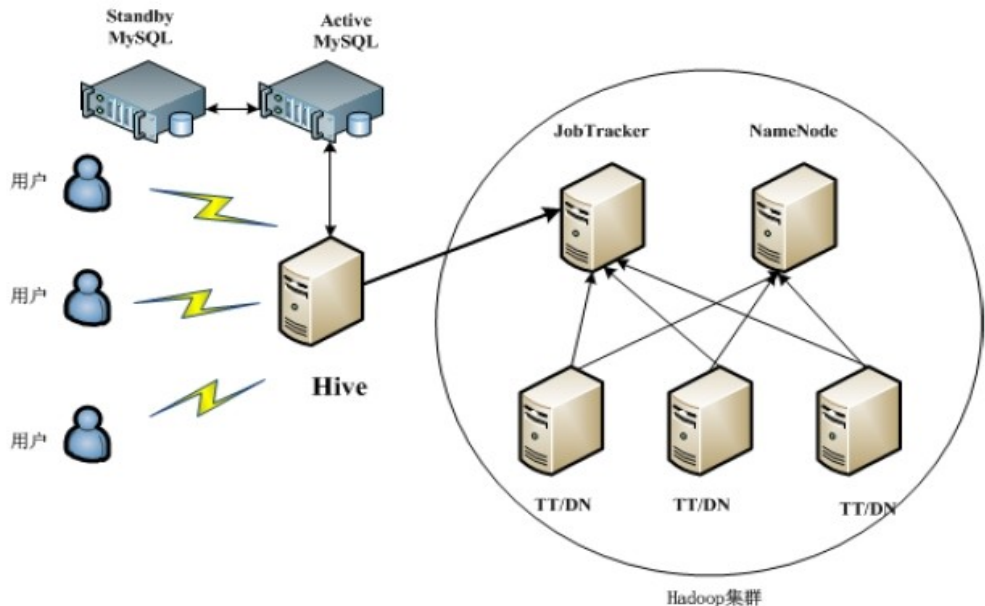
hive 的运行架构



hive 的运行架构

元数据存储 (MetaStore)

- Derby
- MySQL



HIVE元数据表数据字典：

表名	说明
BUCKETING_COLS	Hive表CLUSTERED BY字段信息(字段名，字段序号)
COLUMNS	Hive表字段信息(字段注释，字段名，字段类型，字段序号)
DBS	
NUCLEUS_TABLES	元数据表和hive中class类的对应关系
PARTITIONS	Hive表分区信息(创建时间，具体的分区)
PARTITION_KEYS	Hive分区表分区键(名称，类型，comment，序号)
PARTITION_KEY_VALS	Hive表分区名(键值，序号)
PARTITION_PARAMS	
SDS	所有hive表、表分区所对应的hdfs数据目录和数据格式
SD_PARAMS	
SEQUENCE_TABLE	Hive对象的下一个可用ID
SERDES	Hive表序列化反序列化使用的类库信息
SERDE_PARAMS	序列化反序列化信息，如行分隔符、列分隔符、NULL的表示字符等
SORT_COLS	Hive表SORTED BY字段信息(字段名，sort类型，字段序号)
TABLE_PARAMS	表级属性，如是否外部表，表注释等
TBLS	所有hive表的基本信息

hive 的运行架构

■ 驱动 (Driver)

– 编译器 (hive 的核心)

- 语法解析器 (ParseDriver)
 - 将查询字符串转换成解析树表达式
- 语法分析器 (SemanticAnalyzer)
 - 将解析树转换成基于语句块的内部查询表达式。
- 逻辑计划生成器 (logical plan generator)
 - 将内部查询表达式转换为逻辑计划，这些计划由逻辑操作树组成。
 - 操作符是 hive 的最小处理单元，每个操作符处理代表一道 HDFS 操作或 MR 作业
- 查询计划生成器 (query plan generator)
 - 将逻辑计划转化成物理计划 (MR Task)

– 优化器

- 优化器是一个演化组件。当前，它的规则是：列修剪，谓词下压。

– 执行器

- 编译器将操作树切分为一个 Task 链 (DAG)，执行器会顺序执行其中所有 Task；如果 Task 链 (DAG) 不存在依赖关系时，可采用并发执行的方式进行 Job 的执行



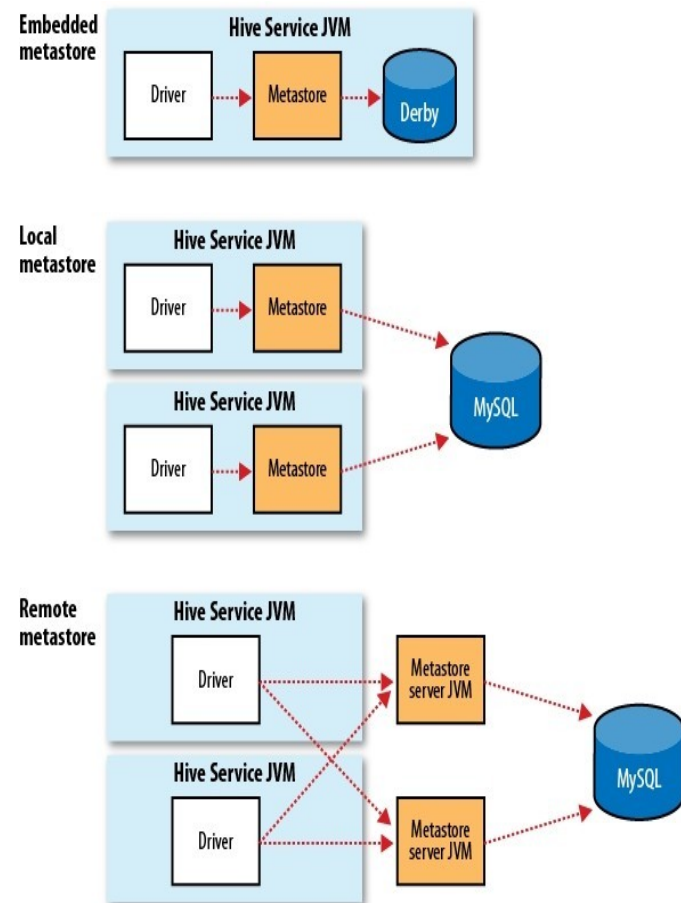
hive 的运行架构

■ 接口

- CLI : 为命令行工具, 为默认服务
 - 启动方式 bin/hive 或 bin/hive --service cli
- hwi : 为 web 接口, 可以通过浏览器访问 hive , 默认端口 9999
 - 启动方式为 bin/hive --service hwi。
- ThriftServer : 通过 Thrift 对外提供服务, 默认端口 10000
 - 启动方式为 bin/hive --service hiveserver。

■ 其他服务 (bin/hive --service -help)

- metastore (bin/hive --service metastore)
- hiveserver2 (bin/hive --service hiveserver2)
 - HiveServer2 是 HiveServer 的改进版本, 它提供新的 Thrift API 来处理 JDBC 或者 ODBC 客户端, Kerberos 身份验证, 多个客户端并发
 - HiveServer2 还提供了新的 CLI : BeeLine , Beeline 是 hive 0.11 引入的新的交互式 CLI , 它基于 SQLLine , 可以作为 Hive JDBC Client 端访问 Hive Server 2 , 启动一个 beeline 就是维护了一个 session。



■ Hadoop

- 用 MapReduce 进行运算
- 存储在 HDFS
 - hive 中所有数据存储在 HDFS 上，包括数据模型中的 Table、Partition、Bucket
 - hive 的默认数据仓库目录是 /user/hive/warehouse，在 hive-site.xml 中由 hive.meta.store.warehouse.dir 项定义
 - 除了 External Table，每个 Table 在数据仓库下都有一个相应的存储目录
 - 当数据被加载至表中时，不会对数据进行任何转换，只是将数据移动到数据仓库目录。
 - Table 被删除时，表数据和元数据都被删除
 - External Table 被删除时，元数据都被删除，表数据不删除
 - 表中的一个 Partition 对应表下的一个子目录
 - 表 log -> /user/hive/warehouse/log
 - log 中含 year 和 month 两个 partition，则：
 - 对于 year=2014，month=6 的子目录为 log/year=2014/month=6
 - 对于 year=2014，month=7 的子目录为 log/year=2014/month=7
 - 每个 Bucket 对应一个文件

hive 的数据模型

- Database
- Table
- Partition
- Bucket
- File



Numeric Types

- **TINYINT** (1-byte signed integer, from -128 to 127)
- **SMALLINT** (2-byte signed integer, from -32,768 to 32,767)
- **INT** (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)
- **BIGINT** (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- **FLOAT** (4-byte single precision floating point number)
- **DOUBLE** (8-byte double precision floating point number)
- **DECIMAL**
 - Introduced in Hive 0.11.0 with a precision of 38 digits
 - Hive 0.13.0 introduced user definable precision and scale

Date/Time Types

- **TIMESTAMP** (Note: Only available starting with Hive 0.8.0)
- **DATE** (Note: Only available starting with Hive 0.12.0)

String Types

- **STRING**
- **VARCHAR** (Note: Only available starting with Hive 0.12.0)
- **CHAR** (Note: Only available starting with Hive 0.13.0)

Misc Types

- **BOOLEAN**
- **BINARY** (Note: Only available starting with Hive 0.8.0)

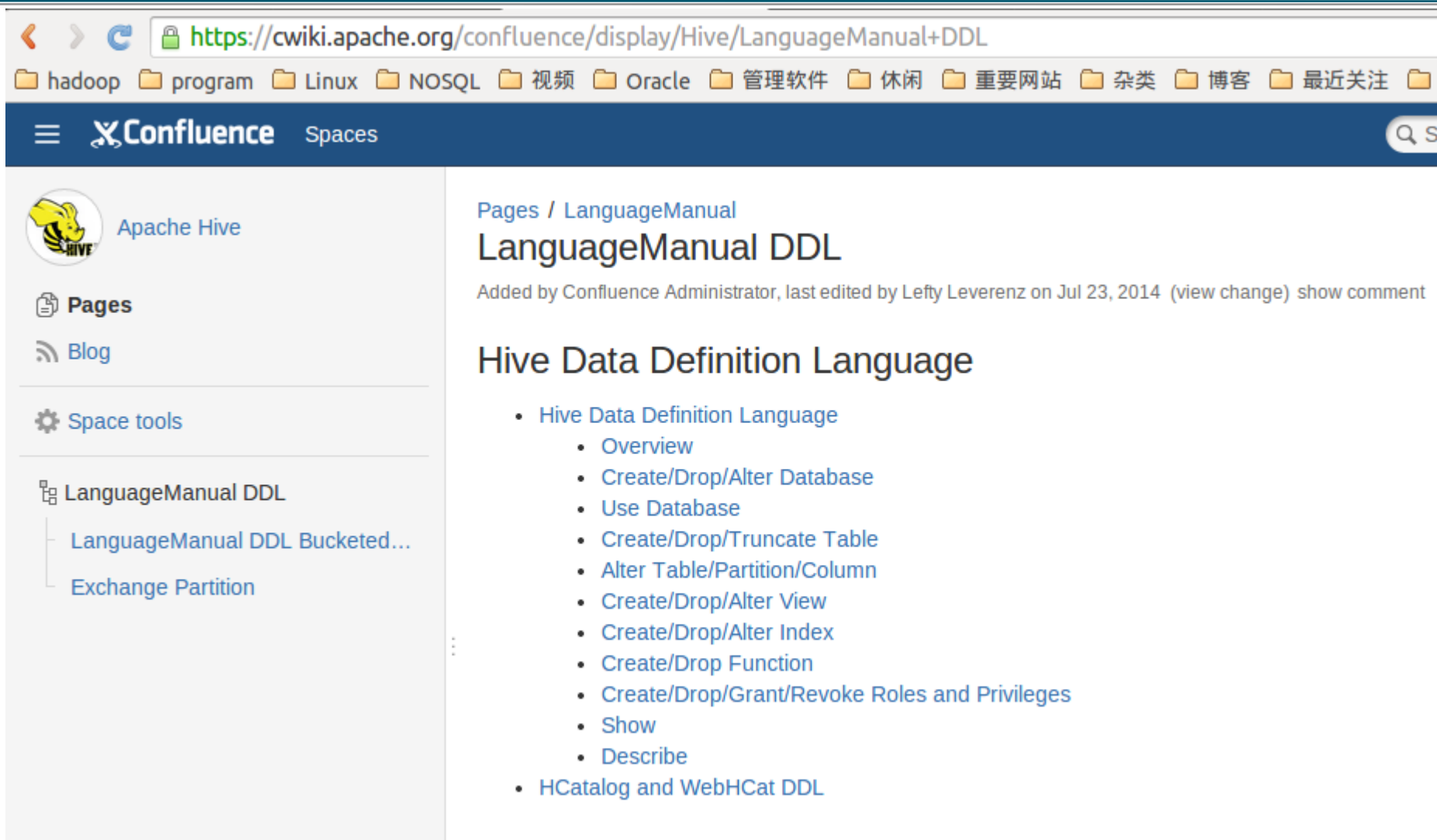


Complex Types

- arrays: ARRAY<data_type>
- maps: MAP<primitive_type, data_type>
- structs: STRUCT<col_name : data_type [COMMENT col_comment], ...>
- union: UNIONTYPE<data_type, data_type, ...> (Note: Only available starting with Hive 0.7.0)

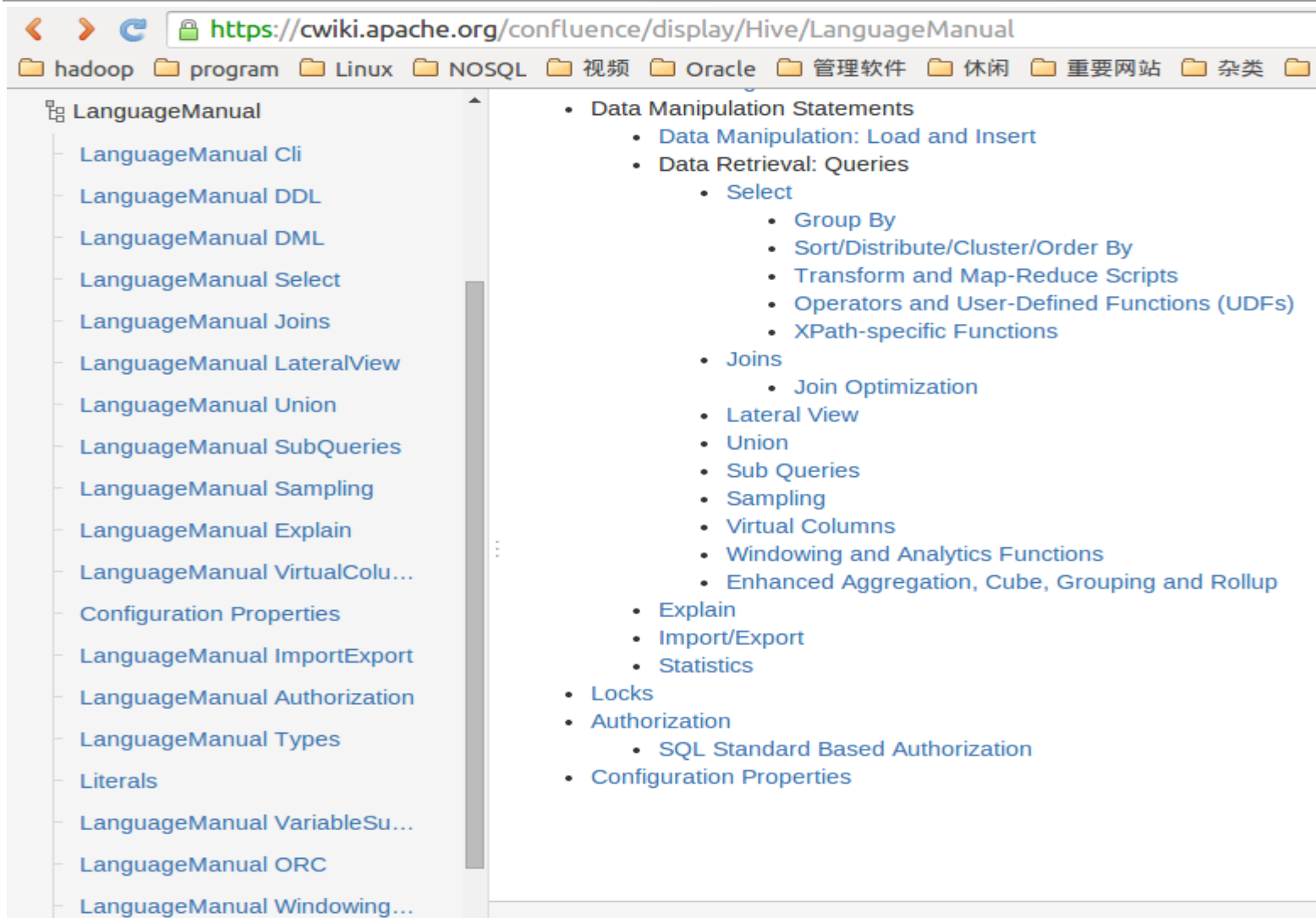
类型	解释	举例
STRUCT	与C/C++中的结构体类似，可通过“.”访问每个域的值，比如STRUCT {first STRING; last STRING}，可通过name.first访问第一个成员。	struct('John', 'Doe')
MAP	存储key/value对，可通过['key']获取每个key的值，比如‘first’→‘John’ and ‘last’→‘Doe’，可通过name['last']获取last name。	map('first', 'John', 'last', 'Doe')
ARRAY	同种类型的数据集合，从0开始索引，比如['John', 'Doe']，可通过name[1]获取“Doe”。	array('John', 'Doe')

对比项目	HQL	SQL
数据插入	支持批量导入	支持单条和批量导入
数据更新	不支持	支持
索引	支持	支持
分区	支持	支持
执行延迟	高	低
扩展性	好	有限



The screenshot shows a web browser displaying the Apache Hive LanguageManual DDL page on Confluence. The browser's address bar shows the URL <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL>. The browser's toolbar includes folders for 'hadoop', 'program', 'Linux', 'NOSQL', '视频', 'Oracle', '管理软件', '休闲', '重要网站', '杂类', '博客', and '最近关注'. The Confluence header features the 'Confluence' logo, 'Spaces', and a search bar. The left sidebar identifies the space as 'Apache Hive' and lists 'Pages' and 'Blog'. Under 'Space tools', it shows a tree view with 'LanguageManual DDL', 'LanguageManual DDL Bucketed...', and 'Exchange Partition'. The main content area is titled 'LanguageManual DDL' and includes a note: 'Added by Confluence Administrator, last edited by Lefty Leverenz on Jul 23, 2014 (view change) show comment'. Below this is the section 'Hive Data Definition Language' with a bulleted list of topics: 'Hive Data Definition Language' (which includes 'Overview', 'Create/Drop/Alter Database', 'Use Database', 'Create/Drop/Truncate Table', 'Alter Table/Partition/Column', 'Create/Drop/Alter View', 'Create/Drop/Alter Index', 'Create/Drop Function', and 'Create/Drop/Grant/Revoke Roles and Privileges'), 'Show', 'Describe', and 'HCatalog and WebHCat DDL'.

hive 的操作



- File Formats
 - Compressed Data Storage
 - LZO Compression
 - Avro
 - ORC Files
 - Parquet

hive 的安装

- 前提：安装和配置 hadoop 集群
- 下载 hive 安装包，解压至安装目录
- 配置 hive-site.xml 和 hive-env.sh
- 配置环境变量
- 启动服务或 cli



- hive 原理和架构
- **hive 演示**

- shark 原理和架构
- shark 演示

- SparkSQL 原理和架构
- SparkSQL 演示

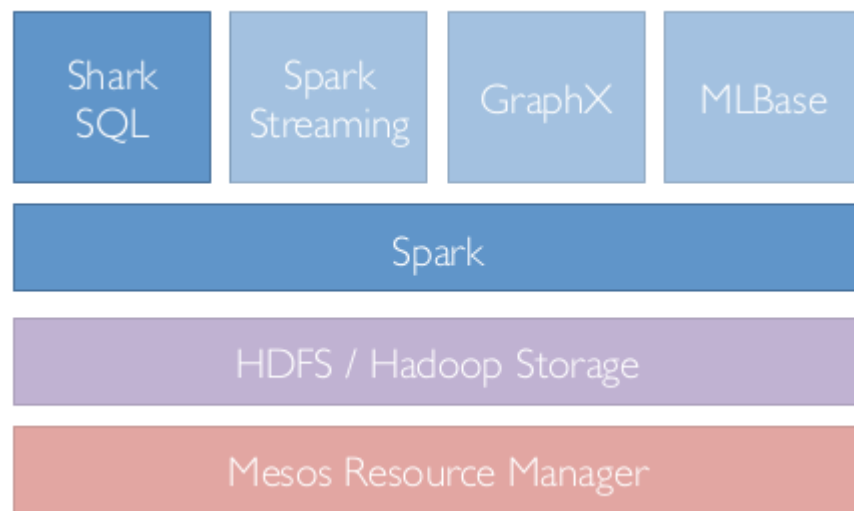


- hive 安装
- 日志数据演示
- 订单交易数据演示



- hive 原理和架构
- hive 演示
- shark 原理和架构
- shark 演示
- SparkSQL 原理和架构
- SparkSQL 演示



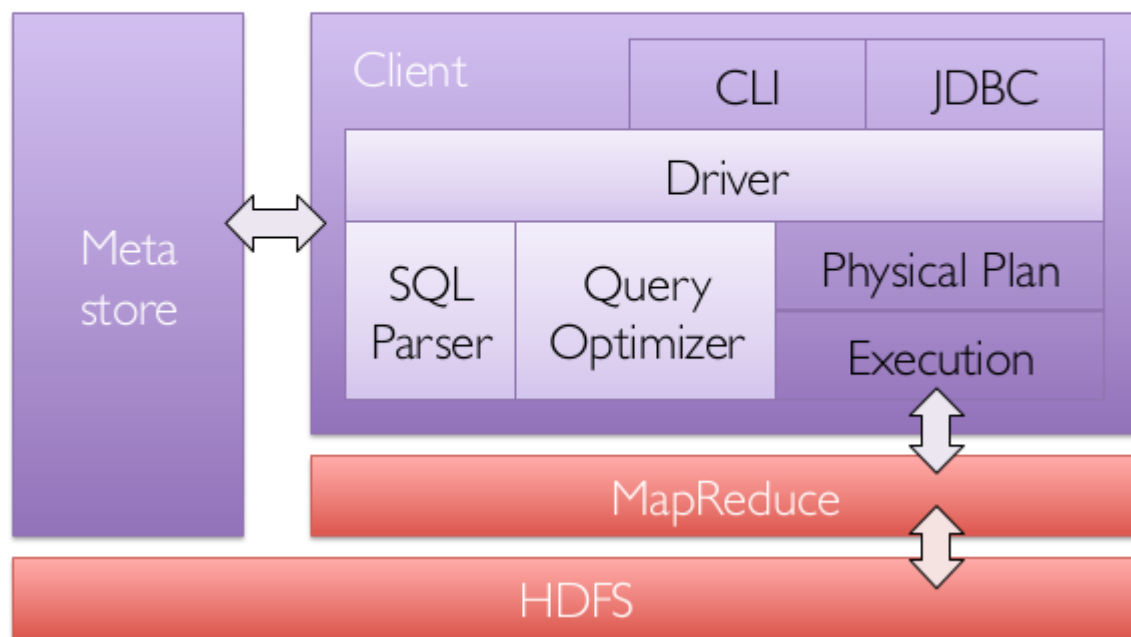


Shark

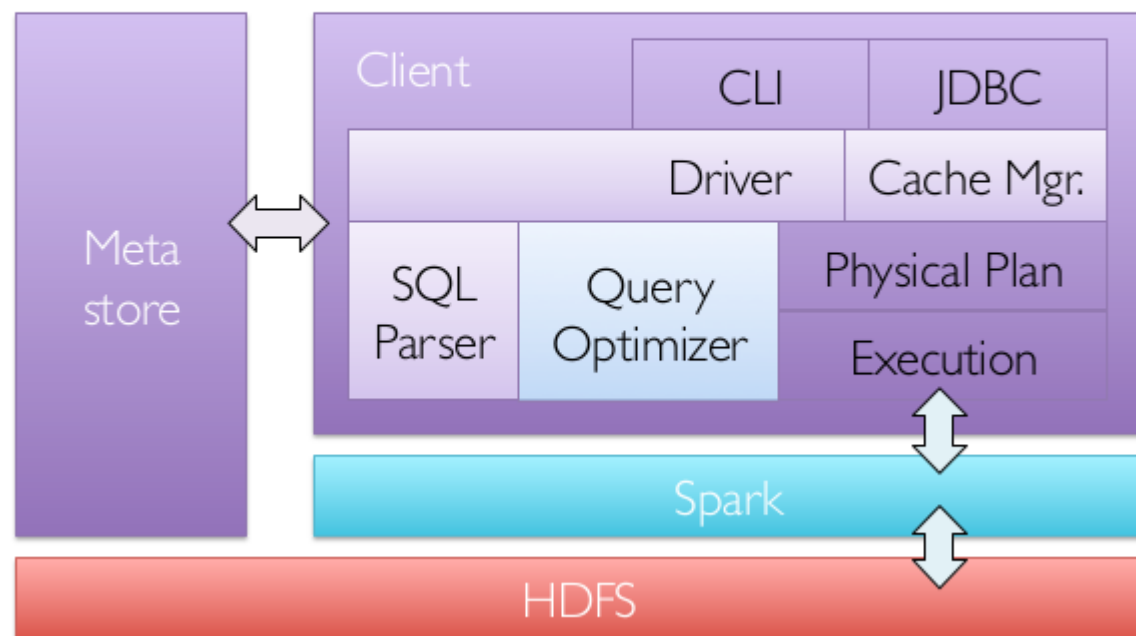
A data analytics system that

- » builds on Spark,
- » scales out and tolerate worker failures,
- » supports low-latency, interactive queries through in-memory computation,
- » supports **both** SQL and complex analytics,
- » is compatible with Hive (storage, serdes, UDFs, types, metadata).

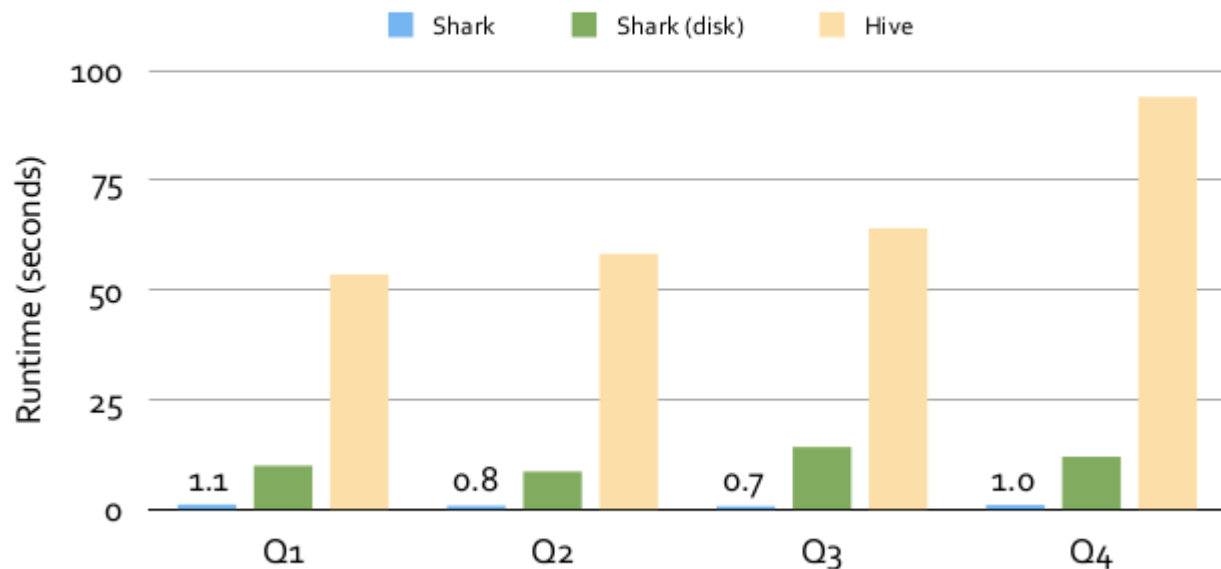
Hive Architecture



Shark Architecture



Performance



1.7 TB Real Warehouse Data on 100 EC2 nodes

Engine Features

Dynamic Query Optimization

Columnar Memory Store

Machine Learning Integration

Data Co-partitioning & Co-location

Partition Pruning based on Range Statistics

...

■ 动态优化 (Dynamic Query Optimization)

- 未发生数据加载过程中的新数据，缺乏数据统计信息；结合普遍使用的 UDF 缺乏，就必须使用动态优化查询。
- 允许基于在运行时收集的数据统计信息来动态优化查询计划

PDE Statistics

1. Gather customizable statistics at per-partition granularities while materializing map output.
 - » partition sizes, record counts (skew detection)
 - » "heavy hitters"
 - » approximate histograms
2. Alter query plan based on such statistics
 - » map join vs shuffle join
 - » symmetric vs non-symmetric hash join
 - » skew handling

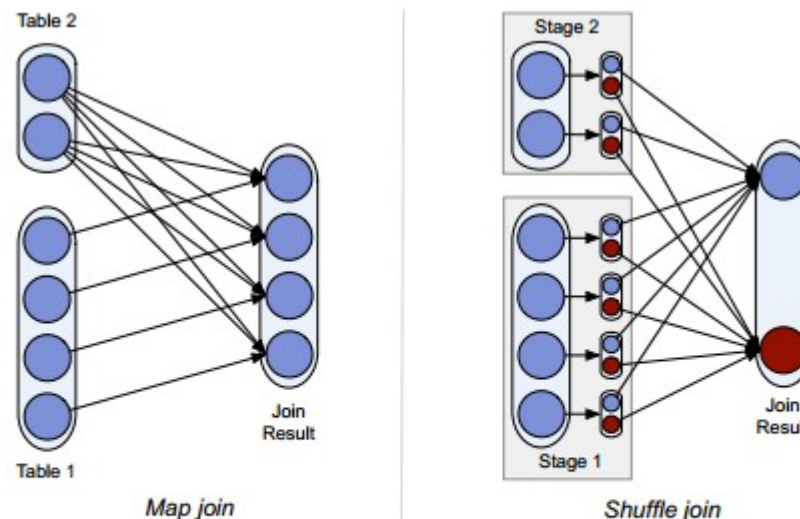
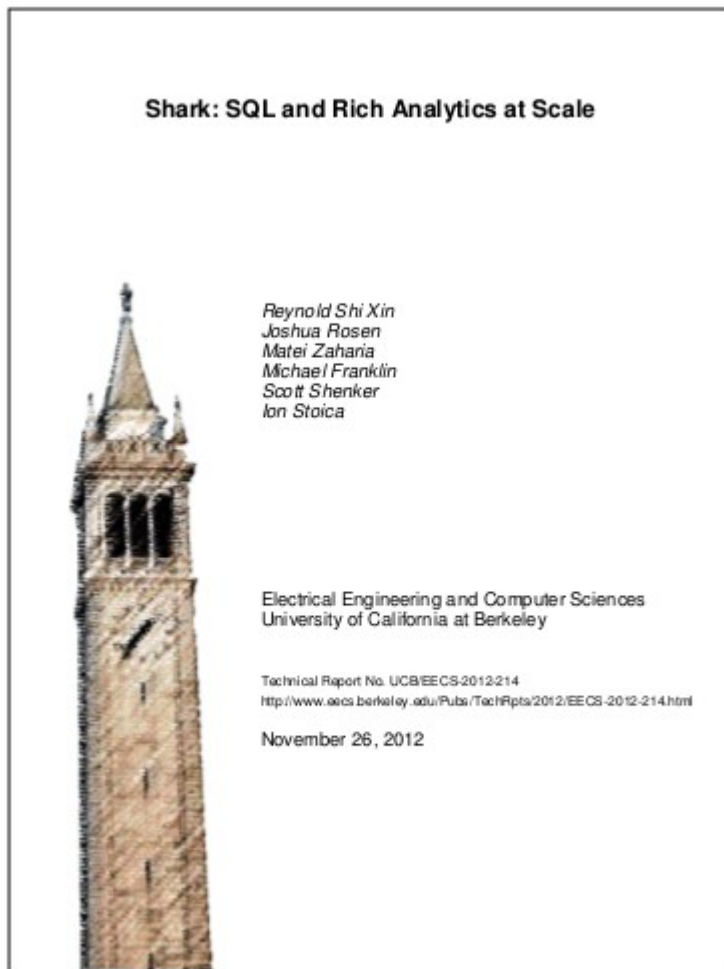


Figure 4: Data flows for map join and shuffle join. Map join broadcasts the small table to all large table partitions, while shuffle join repartitions and shuffles both tables.

- 基于列的压缩和存储 (Columnar Memory Store)
 - Default memory store is to store data partitions as collections of JVM objects.
 - Shark stores all columns of primitive types as JVM primitive arrays.
 - Complex data types supported by Hive, such as map and array, are serialized and concatenated into a single byte array.
 - Each column creates only one JVM object, leading to fast GCs and a compact data representation.
 - The space footprint of columnar data can be further reduced by cheap compression techniques at virtually no CPU cost.
 - Columnar data representation also leads to better cache behavior, especially for analytical queries that frequently compute aggregations on certain columns.



■ More...



- 同 hive 的操作，但要注意所兼容的版本
- 缓存表
 - `CREATE TABLE ... TBLPROPERTIES ("shark.cache" = "true") AS SELECT ...`
 - `CREATE TABLE xxx_cached AS SELECT ...`
- 缓存分区表
 - `CREATE TABLE srcpart_cached (key int, value string) PARTITIONED BY (keypart int);`
 - `shark.cache.policy=shark.memstore2.CacheAllPolicy|LRUCachePolicy|FIFOCachePolicy`
 - `shark.cache.policy.maxSize=100`
 - 可以通过扩展 `shark.memstore2.CachePolicy` 来实现自己的缓存策略
- 当遇到 shuffle data 的时候需要指定 reducer 的数量
 - `set mapred.reduce.tasks = xxx`
- 可以使用 `explain` 看执行计划

shark 的操作

- `./bin/shark` 启动 shark 客户端
- `./bin/shark -e "SELECT * FROM foo"`
- `./bin/shark -i queries.hql` 执行文件中的语句并进入客户端
- `./bin/shark -f queries.hql` 执行文件中的语句
- `./bin/shark -H`

- `sharkserver`
 - `./bin/shark -service sharkserver <port>`
 - `./bin/shark -h <server-host> -p <server-port>`

- 编译 shark
 - `SPARK_HADOOP_VERSION=2.2.0 SPARK_YARN=true sbt/sbt assembly`
 - 更多参数参见 `project/SharkBuild.scala`
- 首先需要将 Shark 与 Spark 一起部署到各个 node(与 hive 不同)
- 环境变量设置 `conf/shark-env.sh`
 - `export JAVA_HOME=`
 - `export SCALA_HOME=`
 - `export HADOOP_HOME=`
 - `export SPARK_HOME=`
 - `export HIVE_HOME=`
 - `export MASTER=`
 - `export SPARK_MEM=`
 - `export SHARK_MASTER_MEM=`
- 你也可以通过 `SPARK_JAVA_OPTS` 来指定相关参数
 - 例如 `SPARK_JAVA_OPTS+="-Dspark.cores.max=16"`

- hive 原理和架构
- hive 演示

- shark 原理和架构
- **shark 演示**

- SparkSQL 原理和架构
- SparkSQL 演示



- shark 安装
- 日志数据演示
- 订单交易数据演示



- hive 原理和架构
- hive 演示

- shark 原理和架构
- shark 演示

- **SparkSQL 原理和架构**
- SparkSQL 演示

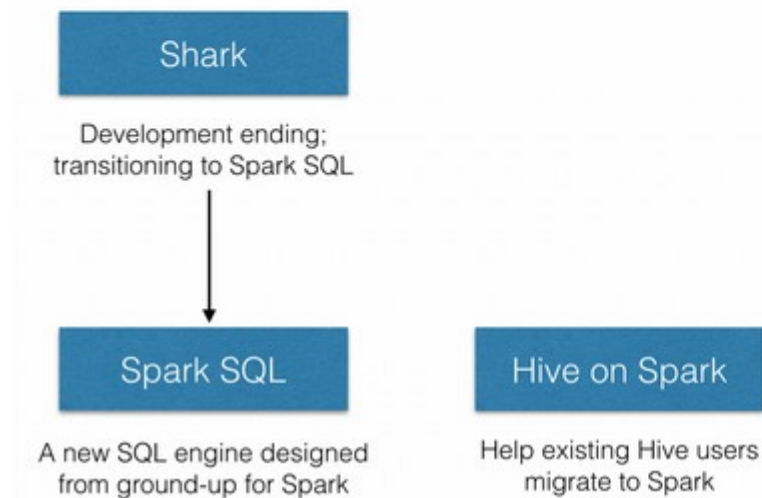


什么是 SparkSQL

July 1, 2014 | by Reynold Xin

Tags: [Spark](#), [Spark SQL](#)

With the introduction of Spark SQL and the new Hive on Spark effort ([HIVE-7292](#)), we get asked a lot about our position in these two projects and how they relate to Shark. At the [Spark Summit](#) today, we announced that we are ending development of Shark and will focus our resources towards Spark SQL, which will provide a superset of Shark's features for existing Shark users to move forward. In particular, Spark SQL will provide both a seamless upgrade path from Shark 0.9 server and new features such as integration with general Spark programs.



Relationship to SHARK

Shark modified the Hive backend to run over Spark, but had two challenges:

- » Limited integration with Spark programs
- » Hive optimizer not designed for Spark

Spark SQL reuses the best parts of Shark:

Borrows

- Hive data loading
- In-memory column store

Adds

- RDD-aware optimizer
- Rich language interfaces

什么是 SparkSQL

Spark SQL at Spark Summit 2014

- 44 contributors
- Alpha release in Spark 1.0
- Support for Hive, Parquet, JSON
- Bindings in Scala, Java and Python
- More exciting features on the horizon!

Spark SQL at Spark Summit 2013

SAN FRANCISCO | DECEMBER 2-3 2013

- 1 developer
- Able to run simple queries over data stored in Hive

Spark SQL Components

运行架构

38%

SQL 操作

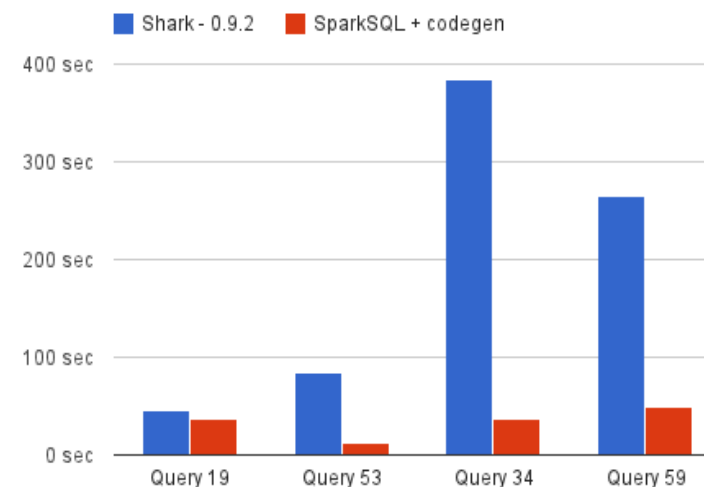
36%

Hive 操作

26%

- Catalyst Optimizer
 - Relational algebra + expressions
 - Query optimization
- Spark SQL Core
 - Execution of queries as RDDs
 - Reading in Parquet, JSON ...
- Hive Support
 - HQL, MetaStore, SerDes, UDFs

TPC-DS Results



- Scala FP features that kill performance:

- Option
- For-loop / map / filter / foreach / ...
- Numeric[T] / Ordering[T] / ...
- Immutable objects (GC stress)



- Have To Resort To:

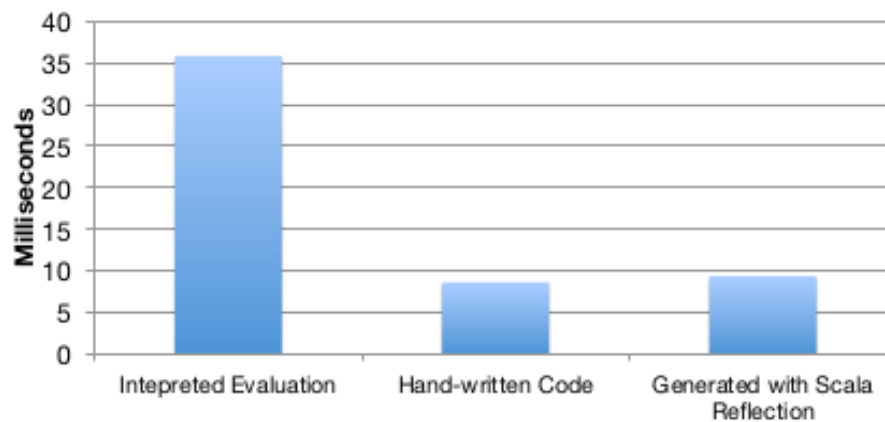
- null
- While-loop and vars
- Manually specialized code for primitive types
- Reusing mutable objects

In-Memory Columnar Storage

Spark SQL can cache tables using an in-memory columnar format

- Scan only required columns
- Fewer allocated objects (values within the same column are serialized into a single byte array to reduce GC)
- Automatically selects best compression

Evaluating 'a+a+a' One Billion Times



什么是 SparkSQL

Code Generation (WIP)

- Generic evaluation of expression logic is very expensive on the JVM
 - Virtual function calls
 - Branches based on expression type
 - Object creation due to primitive boxing
 - Memory consumption by boxed primitive objects
- Generating custom bytecode can eliminate these overheads

Easily Extensible Code Generation

Scala reflection (new in 2.10) makes it easy to extend code generation capabilities:

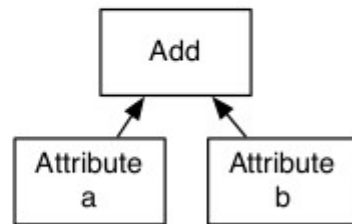
- Pattern matching expressions
- Quasi-quotes turn strings into splice-able Scala ASTs
- Scala does the hard work of generating efficient bytecode.
- Support for generating expression evaluation logic and custom data storage structures.

```
case Cast(e, StringType) =>
  val eval = expressionEvaluator(e)
  eval.code ++
  q"""
    val $nullTerm = ${eval.nullTerm}
    val $primitiveTerm =
      if ($nullTerm)
        ${defaultPrimitive(StringType)}
      else
        ${eval.primitiveTerm}.toString
    """, children
```

SELECT a + b FROM table

Interpreting “a+b”

1. Virtual call to Add.eval()
2. Virtual call to a.eval()
3. Return boxed Int
4. Virtual call to b.eval()
5. Return boxed Int
6. Integer addition
7. Return boxed result



```
def generateCode(e: Expression): Tree = e match {
  case Attribute(ordinal) =>
    q"inputRow.getInt($ordinal)"
  case Add(left, right) =>
    q"""
    {
      val leftResult = ${generateCode(left)}
      val rightResult = ${generateCode(right)}
      leftResult + rightResult
    }
    """, children
}
```

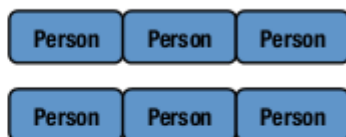
```
val a: Int = inputRow.getInt(0)
val b: Int = inputRow.getInt(1)
val result: Int = a + b
resultRow.setInt(0, result)
```

什么是 SparkSQL

Adding Schema to RDDs

Spark + RDDs

Functional transformations on partitioned collections of **opaque** objects.



SQL + SchemaRDDs

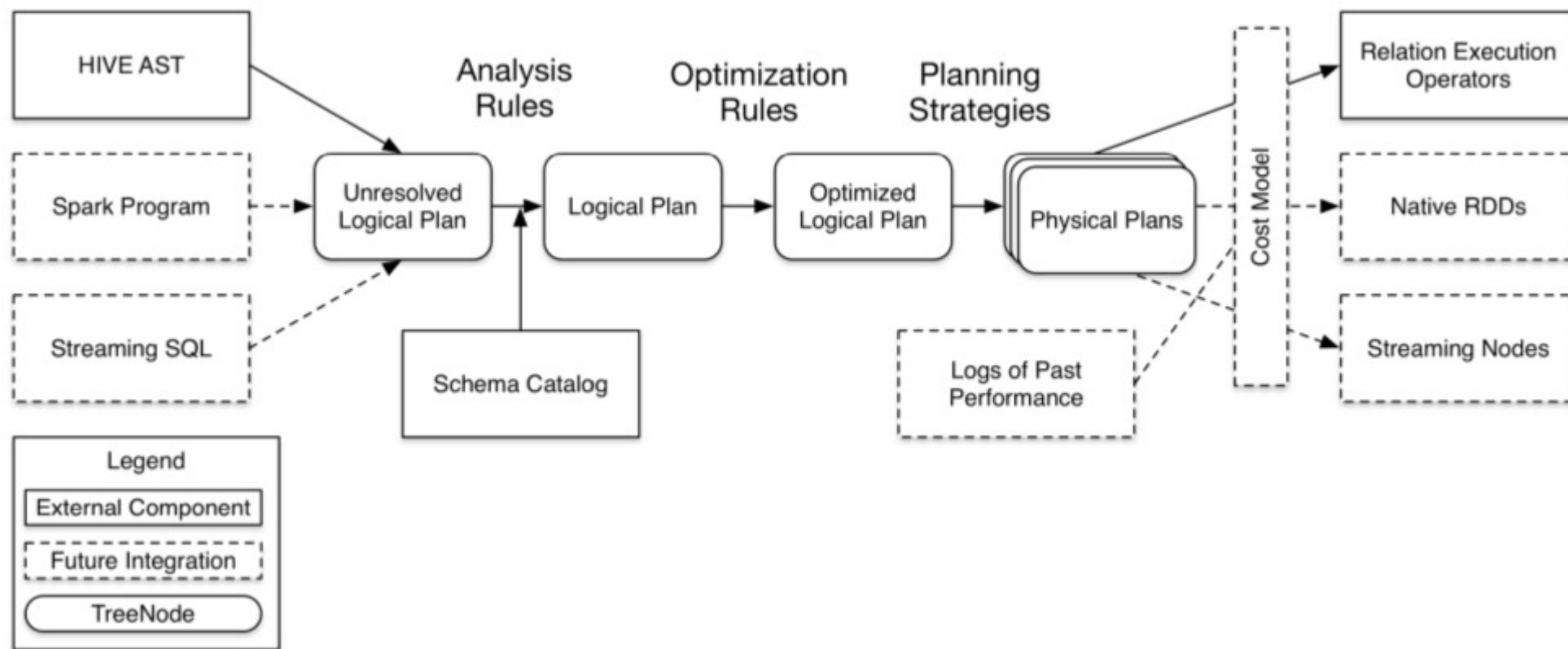
Declarative transformations on partitioned collections of **tuples**.

Name	Age	Height
Name	Age	Height
Name	Age	Height
Name	Age	Height
Name	Age	Height
Name	Age	Height

Unified Data Abstraction

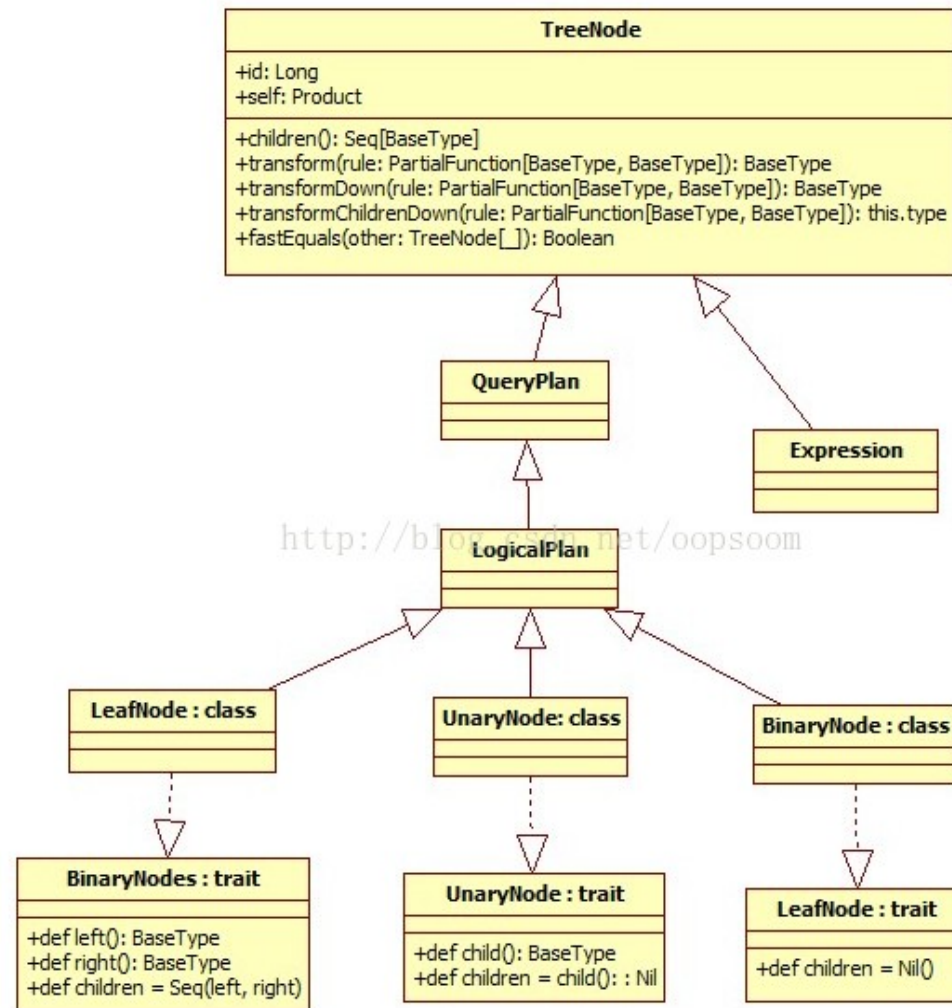


SparkSQL 运行架构



■ TreeNode 体系

- Logical Plans、Expressions、Physical Operators 都可以使用 Tree 表示
- TreeNode 具备一些 scala collection 的操作能力和树遍历能力，树的修改是以替换已有节点的方式进行的。
- TreeNode，内部带一个 children: Seq[BaseType] 表示孩子节点，具备 foreach、map、collect 等针对节点操作的方法，以及 transformDown、transformUp 这样的遍历树上节点，对匹配节点实施变化的方法。
- 三种 trait
 - UnaryNode 一元节点，即只有一个孩子节点。Limit、Filter
 - BinaryNode 二元节点，即有左右孩子的二叉节点。Join、Union
 - LeafNode 叶子节点，没有孩子节点的节点。SetCommand



- Rules 体系
 - 定义策略，通过模式匹配来优化 TreeNode

Rules

- Concise, modular specification of tree transformations
- Can be run once or to fixed point
- Are used for
 - Analysis
 - Providing Hive-specific semantics
 - Optimizations
 - Query planning

Example: Optimization with Rules

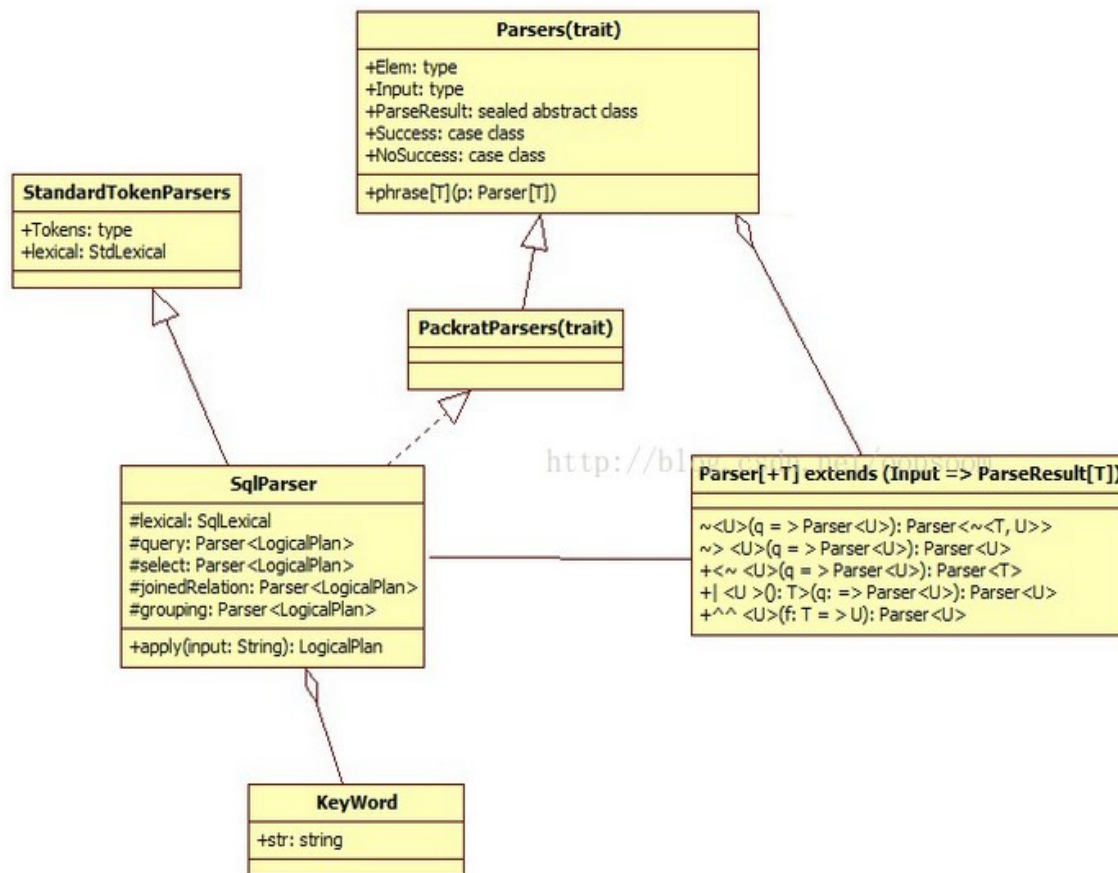
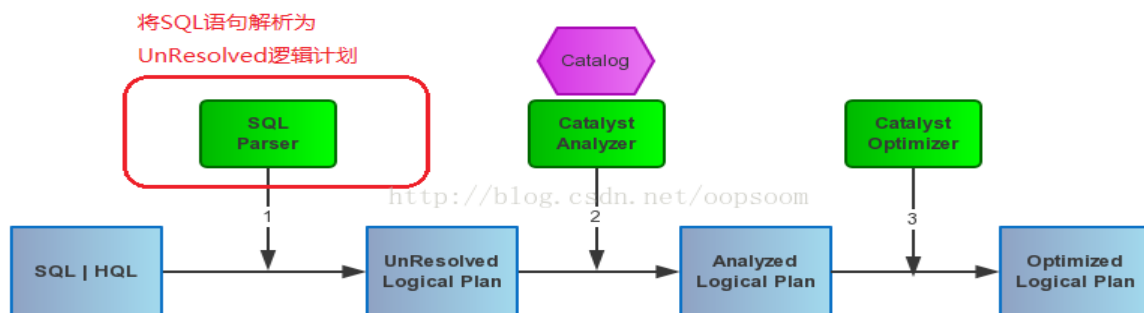
```
object CombineFilters extends Rule[LogicalPlan] {  
  def apply(plan: LogicalPlan): LogicalPlan =  
    plan transform {  
      case Filter(c1, Filter(c2, grandChild)) =>  
        Filter(And(c1,c2), grandChild)  
    }  
}
```

Example: Optimization with Rules

```
object ConstantFolding extends Rule[LogicalPlan] {  
  def apply(plan: LogicalPlan): LogicalPlan =  
    plan transformAllExpression {  
      // Skip redundant folding of literals.  
      case e: Literal => e  
      case e if e.foldable =>  
        Literal(e.apply(EmptyRow), e.dataType)  
    }  
}
```


■ SqlParser

- Sql 解析成 Unresolved 逻辑计划（包含 UnresolvedRelation、UnresolvedFunction、UnresolvedAttribute）
 - 对于命令，会生成一个叶子节点；
 - 对于 SQL 语句，由 lexical 的 Scanner 来扫描输入，分词，校验，如果符合语法就生成 LogicalPlan 语法树，不符合则会提示解析失败。



SparkSQL 运行架构

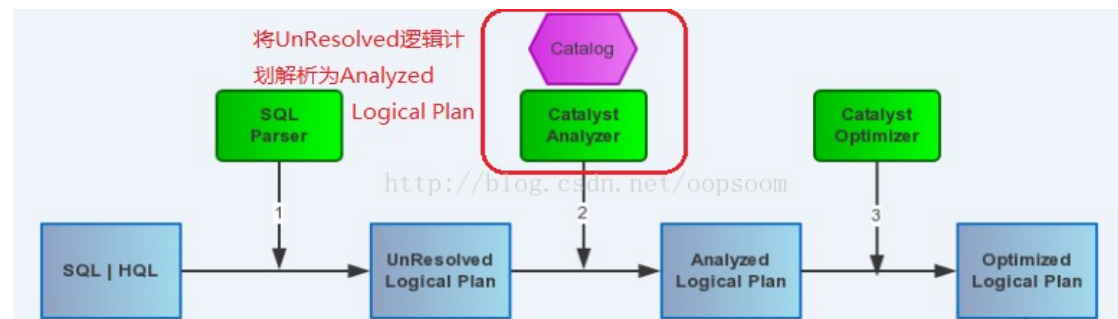
Analyzer

- 流程是实例化一个 SimpleAnalyzer，定义一些 Batch，然后遍历这些 Batch 在 Rule Executor 的环境下，执行 Batch 里面的 Rules，每个 Rule 会对 Unresolved Logical Plan 进行 Resolve，有些可能不能一次解析出，需要多次迭代，直到达到 max 迭代次数或者达到 fix point。
- 比较常用的 Rules 有 ResolveReferences、ResolveRelations、StarExpansion、GlobalAggregates、typeCoercionRules 和 EliminateAnalysisOperators。
- ResolveRelations、ResolveFunctions 等调用了 catalog 这个对象。Catalog 对象里面维护了一个 tableName, Logical Plan 的 HashMap 结果。通过这个 Catalog 目录来寻找当前表的结构，从而从中解析出这个表的字段。

```
class Analyzer(catalog: Catalog, registry: FunctionRegistry, caseSensitive: Boolean)
  extends RuleExecutor[LogicalPlan] with HiveTypeCoercion {

  // TODO: pass this in as a parameter.
  val fixedPoint = FixedPoint(100)

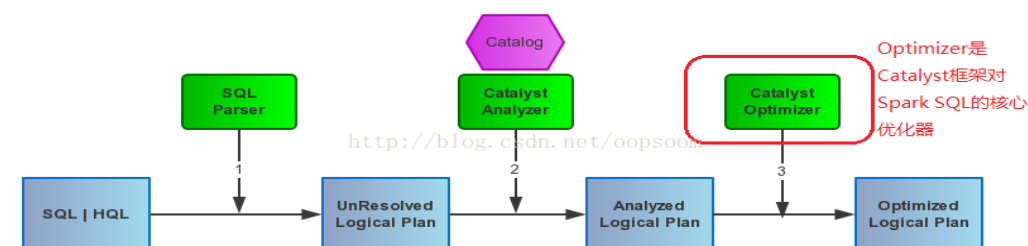
  val batches: Seq[Batch] = Seq(
    Batch("MultiInstanceRelations", Once,
      NewRelationInstances),
    Batch("CaseInsensitiveAttributeReferences", Once,
      (if (caseSensitive) Nil else LowercaseAttributeReferences :: Nil) : _*),
    Batch("Resolution", fixedPoint,
      ResolveReferences ::
      ResolveRelations ::
      NewRelationInstances ::
      ImplicitGenerate ::
      StarExpansion ::
      ResolveFunctions ::
      GlobalAggregates ::
      typeCoercionRules : _*),
    Batch("AnalysisOperators", fixedPoint,
      EliminateAnalysisOperators)
  )
}
```



■ Optimizer

- 将 Analyzed Logical Plan 经过对 Logical Plan 和 Expression 进行 Rule 的应用 transform , 从而达到树的节点进行合并和优化。
- 其中主要的优化的策略总结起来是合并、列裁剪、过滤器下推几大类。
- 对 Logical Plan transform 的是先序遍历 (pre-order) , 而对 Expression transform 的时候是后序遍历 (post-order)
- `val query = sql("select * from (select * from temp_shengli limit 100)a limit 10 ")`
- `val query = sql("select 1+2+3+4 from temp_shengli")`

```
object Optimizer extends RuleExecutor[LogicalPlan] {  
  val batches =  
    Batch("ConstantFolding", FixedPoint(100),  
      NullPropagation,  
      ConstantFolding,  
      BooleanSimplification,  
      SimplifyFilters,  
      SimplifyCasts) ::  
    Batch("Filter Pushdown", FixedPoint(100),  
      CombineFilters,  
      PushPredicateThroughProject,  
      PushPredicateThroughInnerJoin,  
      ColumnPruning) :: Nil  
}
```



■ QueryPlanner

- Abstract class for transforming [[plans.logical.LogicalPlan LogicalPlan]]s into physical plans.
- Child classes are responsible for specifying a list of [[Strategy]] objects that each of which can return a list of possible physical plan options.
- If a given strategy is unable to plan all of the remaining operators in the tree, it can call [[planLater]], which returns a placeholder object that will be filled in using other available strategies

```
abstract class QueryPlanner[PhysicalPlan <: TreeNode[PhysicalPlan]] {  
  /** A list of execution strategies that can be used by the planner */  
  def strategies: Seq[Strategy]  
  
  /**  
   * Given a [[plans.logical.LogicalPlan LogicalPlan]], returns a list of `PhysicalPlan`s that can  
   * be used for execution. If this strategy does not apply to the give logical operation then an  
   * empty list should be returned.  
   */  
  abstract protected class Strategy extends Logging {  
    def apply(plan: LogicalPlan): Seq[PhysicalPlan]  
  }  
  
  /**  
   * Returns a placeholder for a physical plan that executes `plan`. This placeholder will be  
   * filled in automatically by the QueryPlanner using the other execution strategies that are  
   * available.  
   */  
  protected def planLater(plan: LogicalPlan) = apply(plan).next()  
  
  def apply(plan: LogicalPlan): Iterator[PhysicalPlan] = {  
    // Obviously a lot to do here still...  
    val iter = strategies.view.flatMap(_(plan)).toIterator  
    assert(iter.hasNext, s"No plan for $plan")  
    iter  
  }  
}
```

■ 整体运行过程

```
@AlphaComponent
class SQLContext(@transient val sparkContext: SparkContext)
  extends Logging
  with dsl.ExpressionConversions
  with Serializable {

  self =>

  @transient
  protected[sql] lazy val catalog: Catalog = new SimpleCatalog
  @transient
  protected[sql] lazy val analyzer: Analyzer =
    new Analyzer(catalog, EmptyFunctionRegistry, caseSensitive = true)
  @transient
  protected[sql] val optimizer = Optimizer
  @transient
  protected[sql] val parser = new catalyst.SqlParser

  protected[sql] def parseSql(sql: String): LogicalPlan = parser(sql)
  protected[sql] def executeSql(sql: String): this.QueryExecution = executePlan(parseSql(sql))
  protected[sql] def executePlan(plan: LogicalPlan): this.QueryExecution =
    new this.QueryExecution { val logical = plan }
```

■ 整体运行过程

```
protected[sql] class SparkPlanner extends SparkStrategies {...}

@transient
protected[sql] val planner = new SparkPlanner

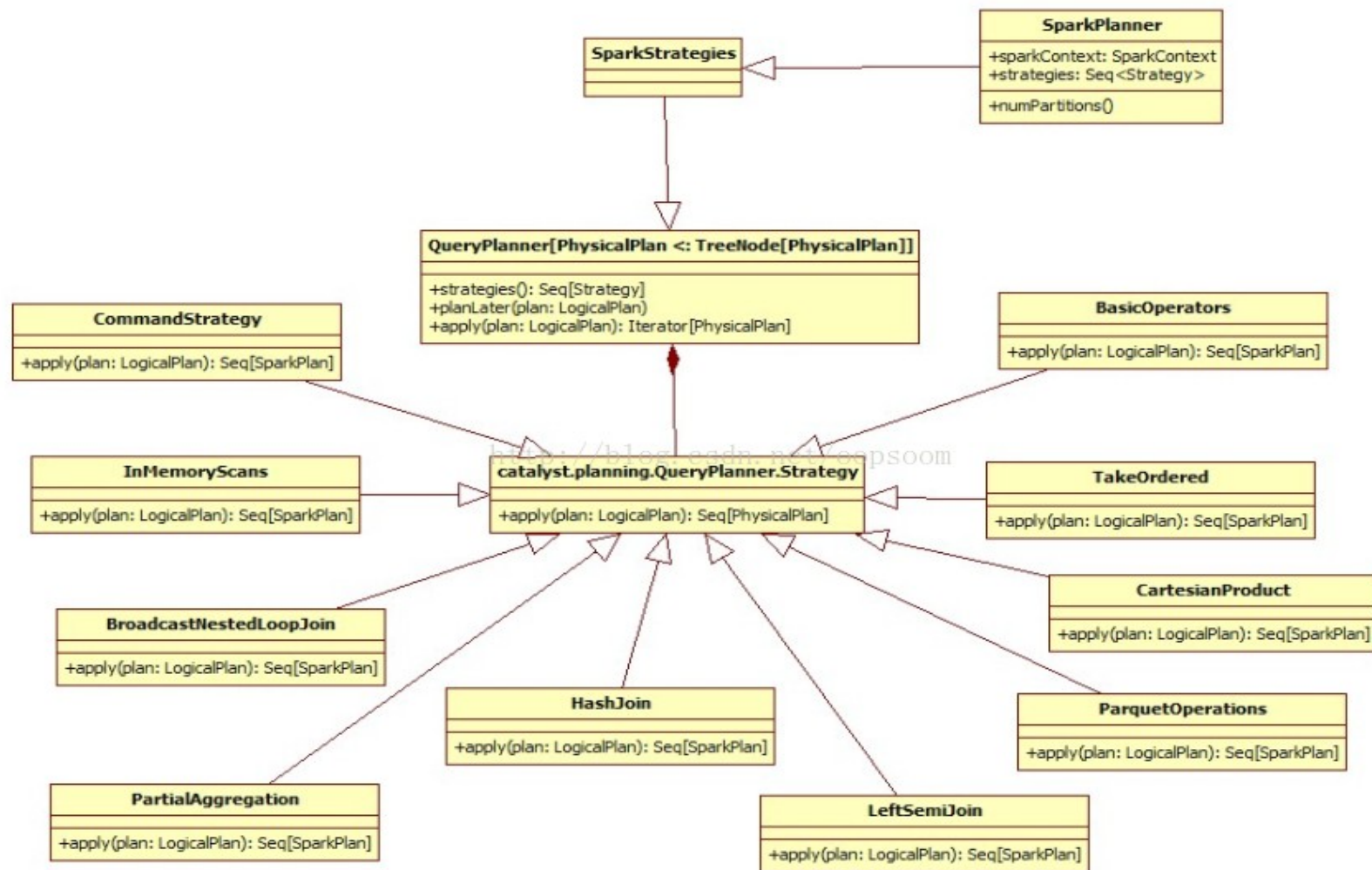
/**
 * Prepares a planned SparkPlan for execution by binding references to specific ordinals, and
 * inserting shuffle operations as needed.
 */
@transient
protected[sql] val prepareForExecution = new RuleExecutor[SparkPlan] {...}

/**
 * The primary workflow for executing relational queries using Spark. Designed to allow easy
 * access to the intermediate phases of query execution for developers.
 */
protected abstract class QueryExecution {
  def logical: LogicalPlan

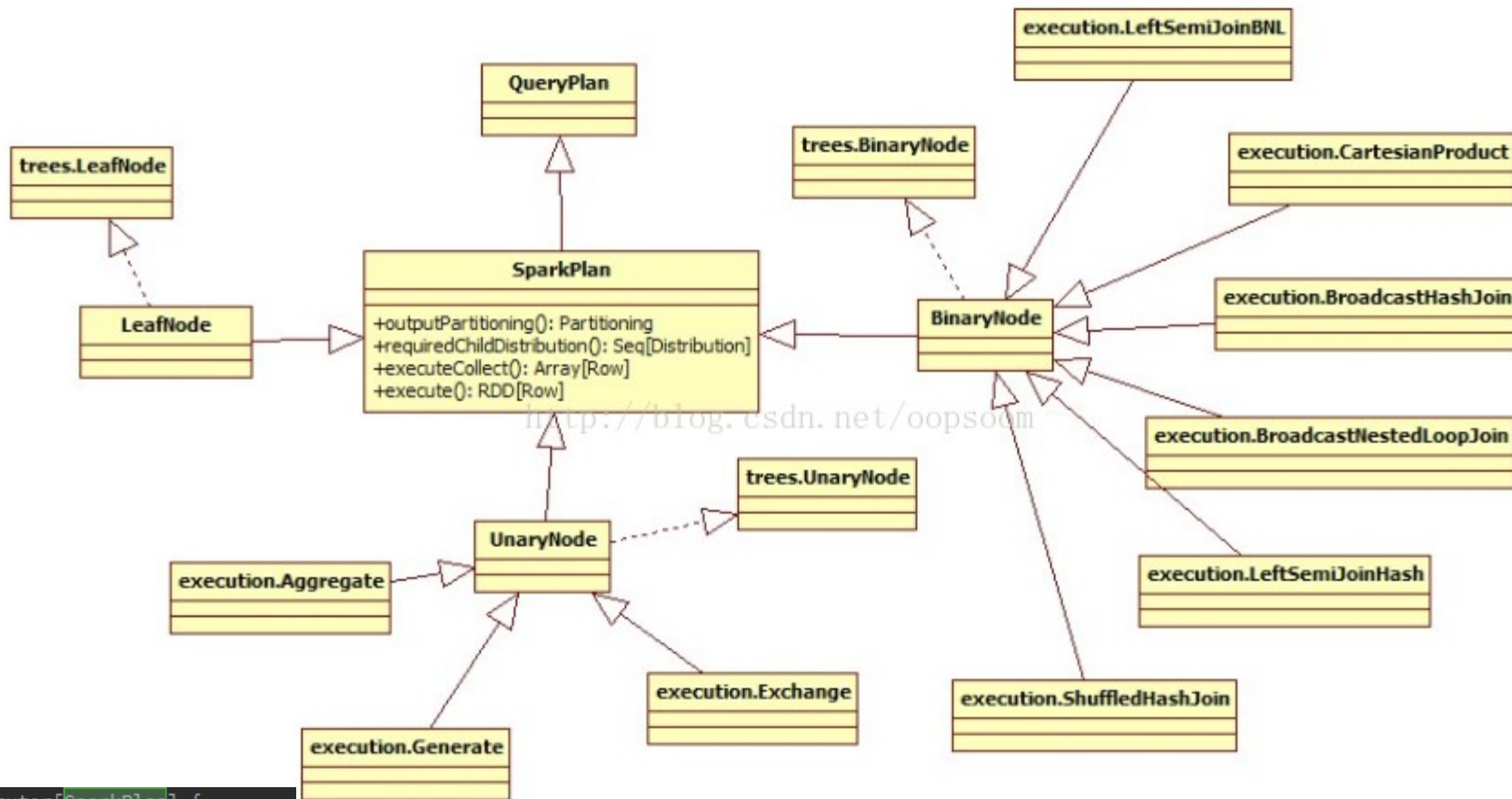
  lazy val analyzed = analyzer(logical)
  lazy val optimizedPlan = optimizer(analyzed)
  // TODO: Don't just pick the first one...
  lazy val sparkPlan = planner(optimizedPlan).next()
  lazy val executedPlan: SparkPlan = prepareForExecution(sparkPlan)

  /** Internal version of the RDD. Avoids copies and has no schema */
  lazy val toRdd: RDD[Row] = executedPlan.execute()
```

整体运行过程

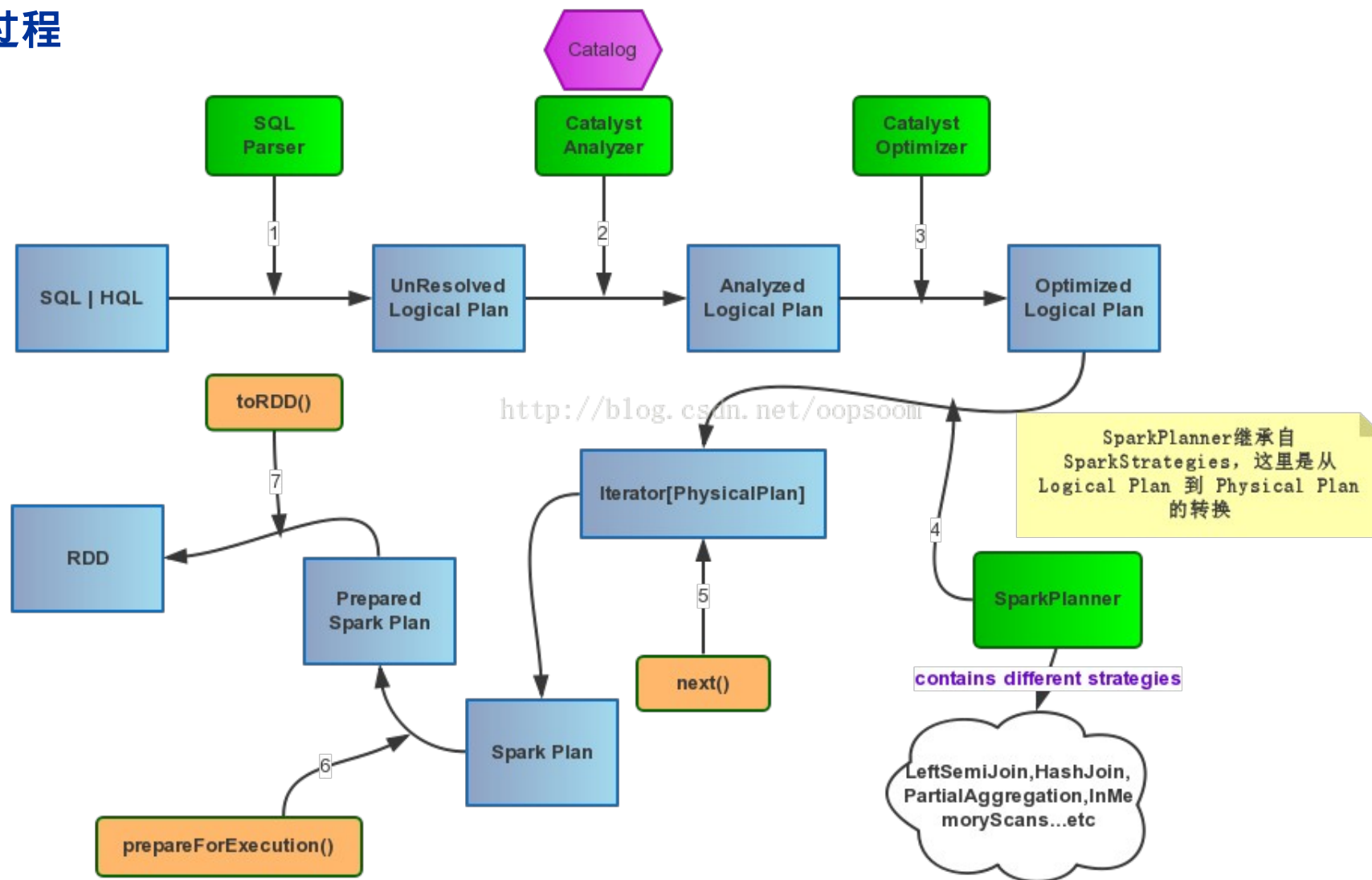


整体运行过程



```
protected[sql] val prepareForExecution = new RuleExecutor[SparkPlan] {
  val batches =
    Batch("Add exchange", Once, AddExchange) ::
    Batch("Prepare Expressions", Once, new BindReferences[SparkPlan]) :: Nil
}
```


■ 整体运行过程



SparkSQL 运行架构

HiveQL

	Spark SQL	Hive Support
Parser 词法语法解析, 产出语法树	Catalyst SqlParser	HiveQl.parseSql
Analyzer 解析出初步的逻辑执行计划	Catalyst Analyzer	Hive MetaStore & Hive UDFs
Optimizer 逻辑执行计划优化	Catalyst Optimizer http://blog.csdn.net/pelick	Catalyst Optimizer
QueryPlanner 逻辑执行计划映射物理执行计划	spark-sql SparkPlanner & SparkStrategy	HivePlanner
Execute 物理执行计划树触发计算	spark-sql SparkPlan.execute()	SparkPlan.execute()

```
/**
 * Returns the AST for the given SQL string.
 */
def getAst(sql: String): ASTNode = ParseUtils.findRootNonNullToken((new ParseDriver).parse(sql))

/** Returns a LogicalPlan for a given HiveQL string. */
def parseSql(sql: String): LogicalPlan = {...}

def parseDdl(ddl: String): Seq[Attribute] = {...}

/** Extractor for matching Hive's AST Tokens. */
object Token {...}
```


RDD 操作

people.txt x

Michael, 29
Andy, 30
Justin, 19

```
14/08/03 14:43:46 INFO SparkContext: Job finished:
Name: Justin
14/08/03 14:43:46 INFO SparkUI: Stopped Spark web UI
```

```
package test

import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext

case class Person(name: String, age: Int)

object sparkSQL_test {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "sparkSQL")
    val sqlContext = new SQLContext(sc)

    import sqlContext._

    val people: RDD[Person] = sc.textFile("file:///home/mmicky/data/spark/people.txt")
      .map(_._split(",")).map(p => Person(p(0), p(1).trim.toInt))
    people.registerAsTable("people")

    val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 10 and age <= 19")
    teenagers.map(t => "Name: " + t(0)).collect().foreach(println)

    sc.stop()
  }
}
```

■ RDD 操作

Relational Queries over RDDs

Spark SQL also supports a DSL (enabled by SchemaRDD) for writing queries. Expressing the query in the last example in this DSL:

```
val teenagers = people  
  .where('age >= 10)  
  .where('age <= 19)  
  .select('name)
```

■ parquet 操作

Parquet Support



```
val sqlContext = new SQLContext(sc)
import sqlContext._

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

val people = sc.textFile("people.txt").map { line =>
  val Array(name, age) = line.split(",")
  Person(name, age.trim.toInt)
}.registerAsTable("people")

people.saveAsParquetFile("people.parquet")
```

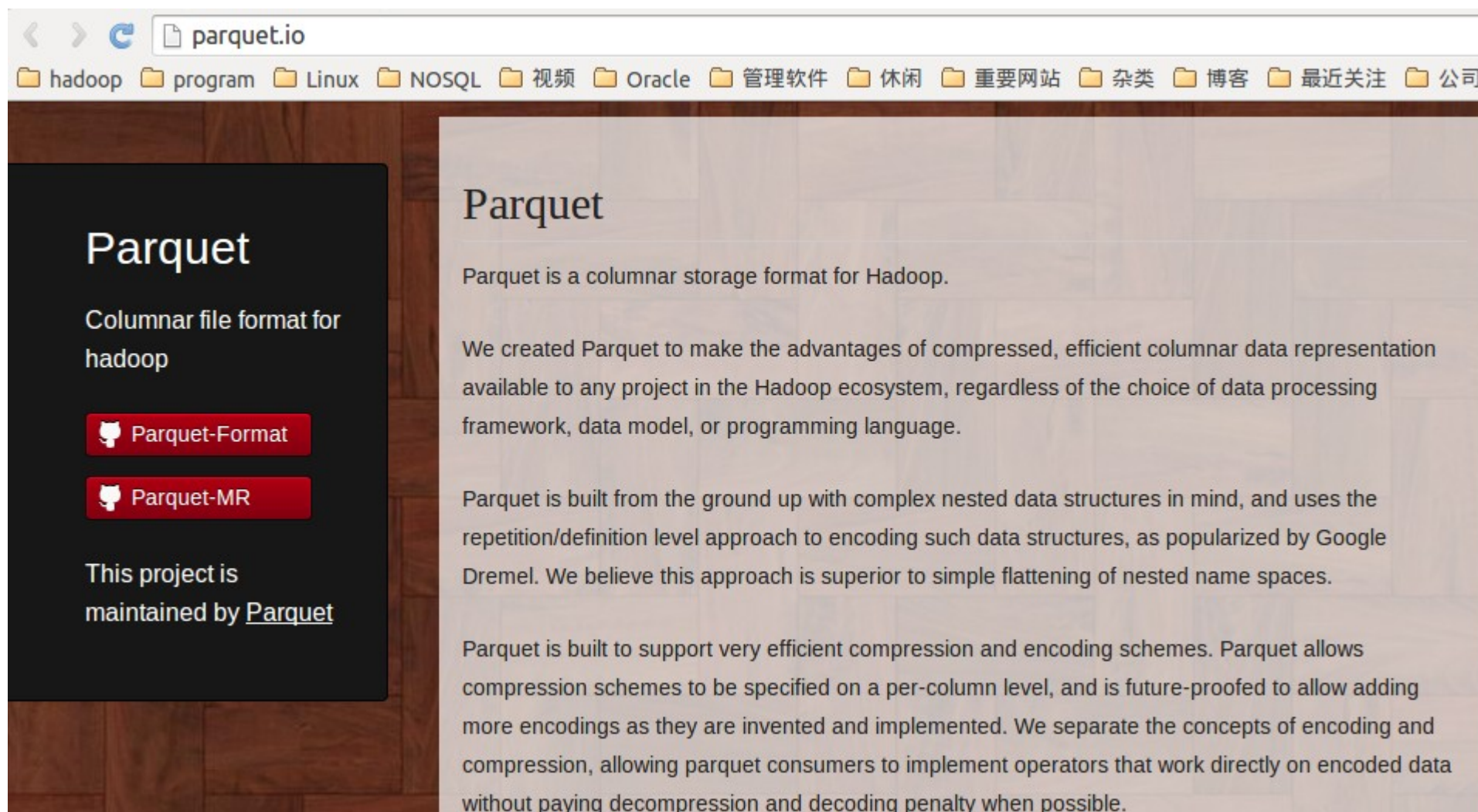
■ parquet 操作

Parquet Support



```
// Parquet files are self-describing so the schema is preserved  
val parquetFile = sqlContext.parquetFile("people.parquet")  
parquetFile.registerAsTable("parquet")  
  
val teenagers = hql(  
  """SELECT name  
    |FROM parquetFile  
    |WHERE age >= 13 AND age <= 19  
  """).stripMargin)  
  
teenagers.collect().foreach(println)
```

■ parquet 操作



hive 操作

Hive Compatibility



- Interfaces that access data and code in the Hive ecosystem
 - Support for writing queries in HiveQL
 - Catalog that interfaces with the Hive MetaStore
 - Table scan operator that uses Hive SerDes
 - Wrappers for Hive UDFs, UDAFs & UDTFs

hive 操作

Hive Compatibility



```
val hiveContext = new HiveContext(sc)
import hiveContext._
```

```
hql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH 'src/main/resources/kv1.txt' INTO TABLE src")
```

```
// Queries are expressed in HiveQL
val schemaRdd = hql("SELECT key, value FROM src")
schemaRdd.collect().foreach(println)
```

■ 混合操作

Hybrid Examples

```
val trainingDataTable = sql(
  """SELECT e.action, u.age, u.latitude, u.longitude
    |FROM Users u JOIN Events e ON u.userId = e.userId
    |""".stripMargin)

// Since `sql` returns an RDD, the results can be easily used in MLlib
val trainingData = trainingDataTable.map {
  case Row(_, age, latitude, longitude) =>
    val features = Array[Double](age, latitude, longitude)
    LabeledPoint(age, features)
}

val model = new LogisticRegressionWithSGD().run(trainingData)
```

Mixing Spark SQL and Machine Learning

■ 混合操作

Hybrid Examples

```
// Data stored in Hive
hql("CREATE TABLE IF NOT EXISTS hiveTable (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH 'kv1.txt' INTO TABLE hiveTable")

// Data in existing RDDs
case class Record(key: Int, value: String)
val rdd = sc.parallelize((1 to 100).map(i => Record(i, s"val_$i")))
rdd.registerAsTable("rddTable")

// Data stored in Parquet
hiveContext.loadParquetFile("parquet.file").registerAsTable("parquetTable")

// Query all sources at once!
hql("SELECT * FROM hiveTable JOIN rddTable JOIN parquetTable WHERE ...")
```

Mixing Multiple Sources

■ 缓存

Caching Tables In-Memory

Spark SQL can cache tables using an in-memory columnar format:

- Scan only required columns
- Fewer allocated objects (less GC)
- Automatically selects best compression

```
cacheTable("people")
```

UDF

Language Integrated UDFs

```
registerFunction("countMatches",  
    lambda (pattern, text):  
        re.subn(pattern, '', text)[1])  
  
sql("SELECT countMatches('a', text)...")
```

- hive 原理和架构
- hive 演示

- shark 原理和架构
- shark 演示

- SparkSQL 原理和架构
- **SparkSQL 演示**



- SQL 演示
 - 文本文件演示
 - Parquet 文件演示
- hive 数据演示
 - 日志数据演示
 - 订单交易数据演示
- 程序编写



- hive
- Shark
- SparkSQL
- 下周预告
 - Spark Streaming 原理和架构
 - Spark 实例演示

See You Next



Thanks

FAQ 时间