Parallel Processing
Spark and Spark SQL

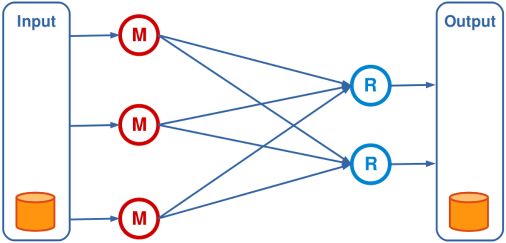
Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology

Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.

- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.

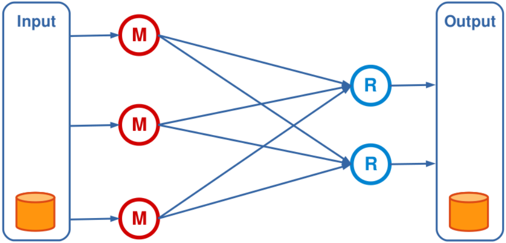


Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.

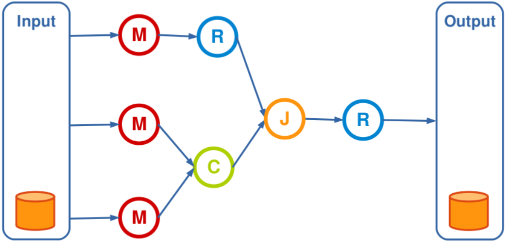
- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.

- MapReduce greatly simplified big data analysis on large unreliable clusters.



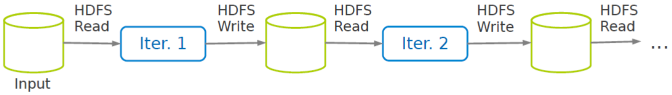
Motivation (2/4)

- MapReduce programming model has not been designed for complex operations, e.g., data mining.



Motivation (3/4)

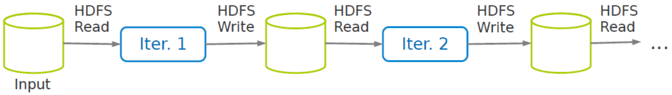
- Very expensive (slow), i.e., always goes to disk and HDFS.



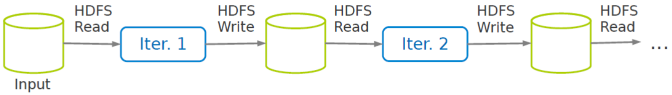
Motivation (4/4)

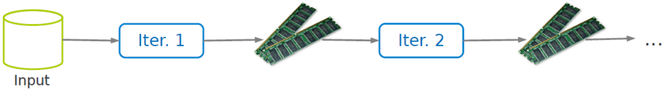
- Extends MapReduce with more operators.
- Support for advanced data flow graphs.
- In-memory and out-of-core processing.

Spark vs. MapReduce (1/2)

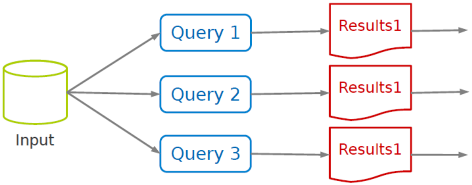


Spark vs. MapReduce (1/2)

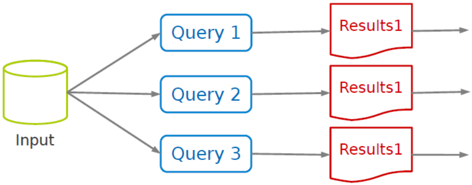


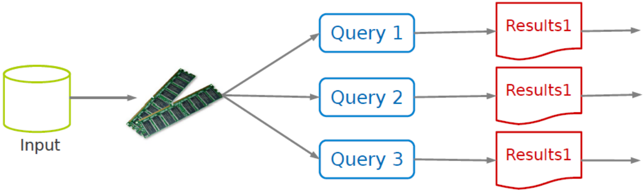


Spark vs. MapReduce (2/2)



Spark vs. MapReduce (2/2)





Challenge

How to design a distributed memory abstraction that is both fault tolerant and efficient?

Challenge

How to design a distributed memory abstraction that is both fault tolerant and efficient?

Solution

Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets (RDD) (1/2)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.
  - Like a `LinkedList <MyObjects>`

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.
- Built through coarse grained transformations, e.g., `map`, `filter`, `join`.

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.
- Built through coarse grained transformations, e.g., `map`, `filter`, `join`.
- Fault tolerance via automatic rebuild (no replication).

RDD Applications

- Applications suitable for RDDs
  - Batch applications that apply the same operation to all elements of a dataset.

- Applications not suitable for RDDs
  - Applications that make asynchronous fine-grained updates to shared state, e.g., storage system for a web application.

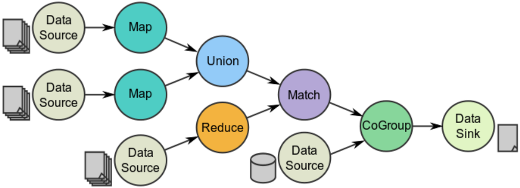
Programming Model

Spark Programming Model (1/2)

- Spark programming model is based on parallelizable operators.
- Parallelizable operators are higher-order functions that execute user-defined functions in parallel.

Spark Programming Model (2/2)

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Job description based on directed acyclic graphs (DAG).



Higher-Order Functions (1/3)

- Higher-order functions: RDDs operators.
- There are two types of RDD operators: transformations and actions.

Higher-Order Functions (2/3)

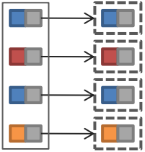
- Transformations: lazy operators that create new RDDs.
- Actions: lunch a computation and return a value to the program or write data to the external storage.

Higher-Order Functions (3/3)

Transformations	$map(f: \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f: \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f: \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction: \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f: (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f: \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c: \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p: \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f: (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k: \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path: \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

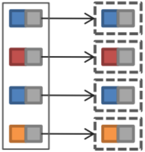
RDD Transformations - Map

- All pairs are independently processed.



RDD Transformations - Map

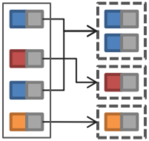
- All pairs are independently processed.



```scala
// passing each element through a function.
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x) // {1, 4, 9}

// selecting those elements that func returns true.
val even = squares.filter(x => x % 2 == 0) // {4}

// mapping each element to zero or more others.
nums.flatMap(x => Range(0, x, 1)) // {0, 0, 1, 0, 1, 2}
```

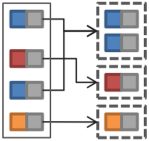
RDD Transformations - Reduce

- Pairs with identical key are grouped.
- Groups are independently processed.



RDD Transformations - Reduce

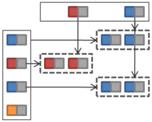
- Pairs with identical key are grouped.
- Groups are independently processed.



```
val pets = sc.parallelize(Seq(("cat", 1), ("dog", 1), ("cat", 2)))

pets.groupByKey()
// {(cat, (1, 2)), (dog, (1))}

pets.reduceByKey((x, y) => x + y)
// {(cat, 3), (dog, 1)}
```

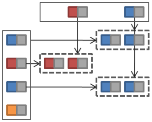
RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



```
val visits = sc.parallelize(Seq(("index.html", "1.2.3.4"),
                                ("about.html", "3.4.5.6"),
                                ("index.html", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("index.html", "Home"),
                                   ("about.html", "About")))

visits.join(pageNames)
  // ("index.html", ("1.2.3.4", "Home"))
  // ("index.html", ("1.3.3.1", "Home"))
  // ("about.html", ("3.4.5.6", "About"))
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```scala
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Return the number of elements in the RDD.

```
nums.count() // 3
```

Basic RDD Actions (2/2)

- Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y)
or
nums.reduce(_ + _) // 6
```

Basic RDD Actions (2/2)

- Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y)
or
nums.reduce(_ + _) // 6
```

- Write the elements of the RDD as a text file.

```
nums.saveAsTextFile("hdfs://file.txt")
```

SparkContext

- Main entry point to Spark functionality.
- Available in shell as variable sc.
- Only one SparkContext may be active per JVM.

```
// master: the master URL to connect to, e.g.,
// "local", "local[4]", "spark://master:7077"
val conf = new SparkConf().setAppName(appName).setMaster(master)

new SparkContext(conf)
```

Creating RDDs

- Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

Creating RDDs

- Turn a collection into an RDD.

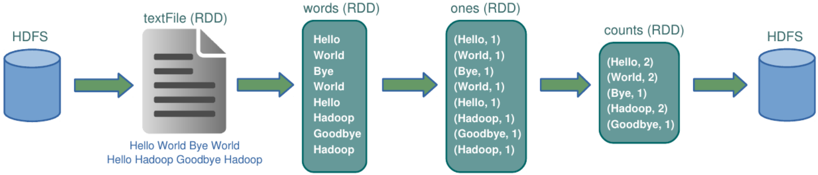
```
val a = sc.parallelize(Array(1, 2, 3))
```

- Load text file from local FS, HDFS, or S3.

```
val a = sc.textFile("file.txt")
val b = sc.textFile("directory/*.txt")
val c = sc.textFile("hdfs://namenode:9000/path/file")
```

Example 1

```
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



Example 2

```
val textFile = sc.textFile("hdfs://...")
val sics = textFile.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_ + _)
```

Example 2

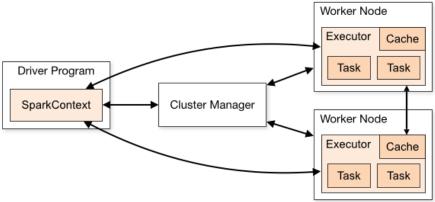
```
val textFile = sc.textFile("hdfs://...")
val sics = textFile.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_ + _)
```

```
val textFile = sc.textFile("hdfs://...")
val count = textFile.filter(_.contains("SICS")).count()
```

Execution Engine

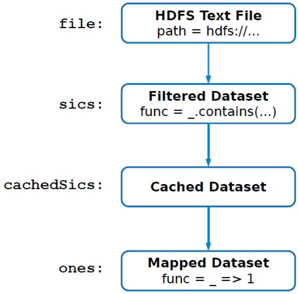
Spark Programming Interface

- A Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster.



Lineage

- Lineage: transformations used to build an RDD.
- RDDs are stored as a chain of objects capturing the lineage of each RDD.

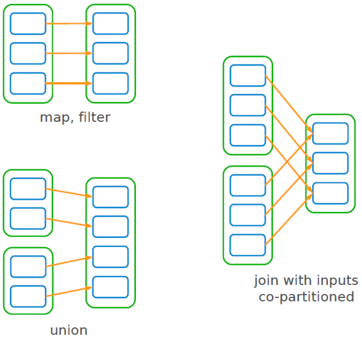



```
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

RDD Dependencies (1/3)

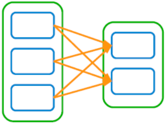
- Two types of dependencies between RDDs: Narrow and Wide.

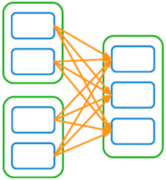
RDD Dependencies: Narrow (2/3)



- Narrow: each partition of a parent RDD is used by at most one partition of the child RDD.
- Narrow dependencies allow pipelined execution on one cluster node, e.g., a `map` followed by a `filter`.

RDD Dependencies: Wide (3/3)





- Wide: each partition of a parent RDD is used by multiple partitions of the child RDDs.

Job Scheduling (1/3)

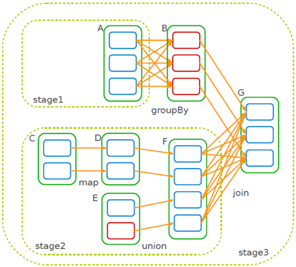
- Similar to Dryad.
- But, it takes into account which partitions of persistent RDDs are available in memory.

Job Scheduling (2/3)

- When a user runs an action on an RDD: the scheduler builds a DAG of stages from the RDD lineage graph.

- A stage contains as many pipelined transformations with narrow dependencies.

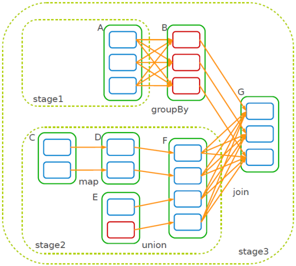
- The boundary of a stage:
  - Shuffles for wide dependencies.
  - Already computed partitions.



Job Scheduling (3/3)

- The scheduler launches tasks to compute missing partitions from each stage until it computes the target RDD.

- Tasks are assigned to machines based on data locality.
    - If a task needs a partition, which is available in the memory of a node, the task is sent to that node.



RDD Fault Tolerance (1/2)

- RDDs maintain lineage information that can be used to reconstruct lost partitions.
- Logging lineage rather than the actual data.
- No replication.
- Recompute only the lost partitions of an RDD.

RDD Fault Tolerance (2/2)

- The intermediate records of wide dependencies are materialized on the nodes holding the parent partitions: to simplify fault recovery.

- If a task fails, it will be re-ran on another node, as long as its stages parents are available.

- If some stages become unavailable, the tasks are submitted to compute the missing partitions in parallel.

Memory Management (1/2)

- If there is not enough space in memory for a new computed RDD partition: a partition from the least recently used RDD is evicted.

- Spark provides three options for storage of persistent RDDs:
  1. In memory storage as deserialized Java objects.
  2. In memory storage as serialized Java objects.
  3. On disk storage.

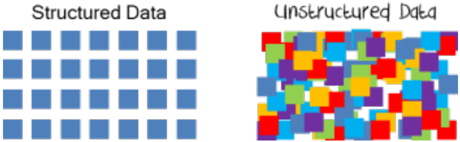
Memory Management (2/2)

- When an RDD is persisted, each node stores any partitions of the RDD that it computes in memory.
- This allows future actions to be much faster.
- Persisting an RDD using `persist()` or `cache()` methods.

Structured Data Processing

Motivation

- Users often prefer writing declarative queries.
- Lack of schema.



Hive

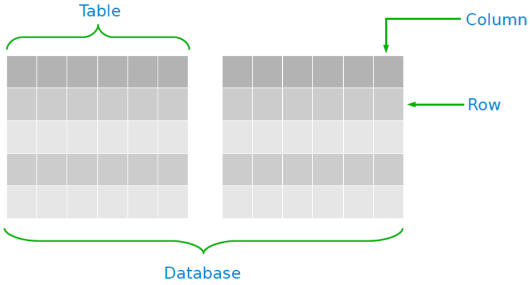
- A system for managing and querying structured data built on top of MapReduce.

- Converts a query to a series of MapReduce phases.

- Initially developed by Facebook.

Hive Data Model

- Re-used from RDBMS:
  - Database: Set of Tables.
  - Table: Set of Rows that have the same schema (same columns).
  - Row: A single record; a set of columns.
  - Column: provides value and type for a single value.



Hive API (1/2)

- HiveQL: SQL-like query languages

Hive API (1/2)

- HiveQL: SQL-like query languages
- DDL operations (Data Definition Language)
  - Create, Alter, Drop

Hive API (1/2)

- HiveQL: SQL-like query languages
- DDL operations (Data Definition Language)
  - Create, Alter, Drop
- DML operations (Data Manipulation Language)
  - Load and Insert (overwrite)
  - Does not support updating and deleting

Hive API (1/2)

- HiveQL: SQL-like query languages
- DDL operations (Data Definition Language)
  - Create, Alter, Drop
- DML operations (Data Manipulation Language)
  - Load and Insert (overwrite)
  - Does not support updating and deleting
- Query operations
  - Select, Filter, Join, Groupby

Hive API (2/2)

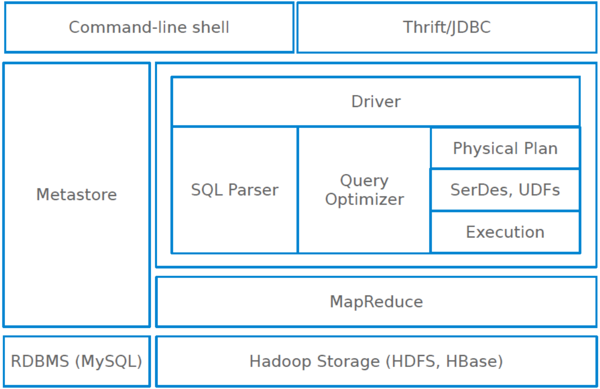
```
-- DDL: creating a table with three columns
CREATE TABLE customer (id INT, name STRING, address STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

-- DML: loading data from a flat file
LOAD DATA LOCAL INPATH 'data.txt' OVERWRITE INTO TABLE customer;

-- Query: joining two tables
SELECT * FROM customer c JOIN order o ON (c.id = o.cus_id);
```

Executing SQL Questions

- Processes HiveQL statements and generates the execution plan through three-phase processes.

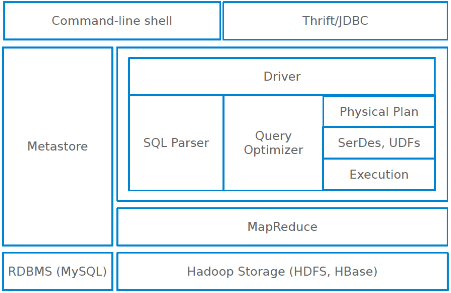
1. Query parsing: transforms a query string to a parse tree representation.
2. Logical plan generation: converts the internal query representation to a logical plan, and optimizes it.
3. Physical plan generation: split the optimized logical plan into multiple map/reduce and HDFS tasks.

Hive Components (1/8)



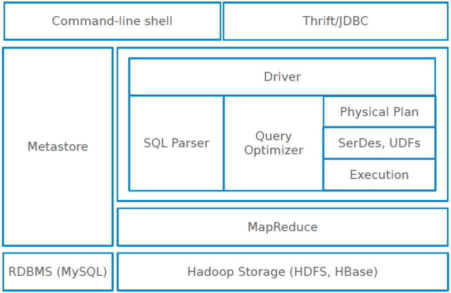
Hive Components (2/8)

- External interfaces
  - User interfaces, e.g., CLI and web UI
  - Application programming interfaces, e.g., JDBC and ODBC
  - Thrift, a framework for cross-language services.



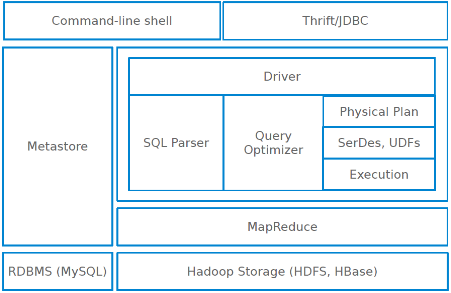
Hive Components (3/8)

- Driver
  - Manages the life cycle of a HiveQL statement during compilation, optimization and execution.



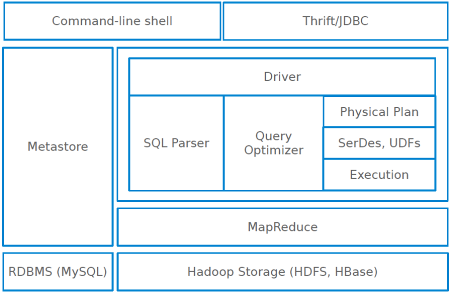
Hive Components (4/8)

- Compiler (Parser/Query Optimizer)
  - Translates the HiveQL statement into a a logical plan, and optimizes it.



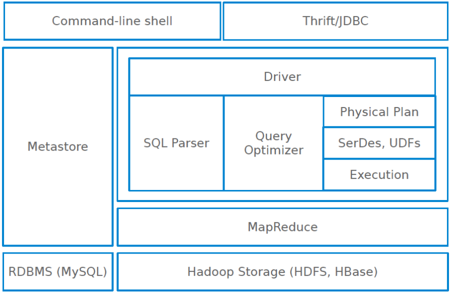
Hive Components (5/8)

- Physical plan
  - Transforms the logical plan into a DAG of Map/Reduce jobs.



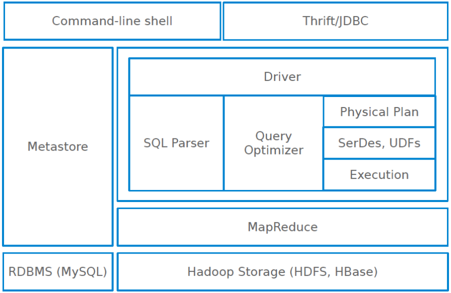
Hive Components (6/8)

- Execution engine
  - The driver submits the individual mapreduce jobs from the DAG to the execution engine in a topological order.



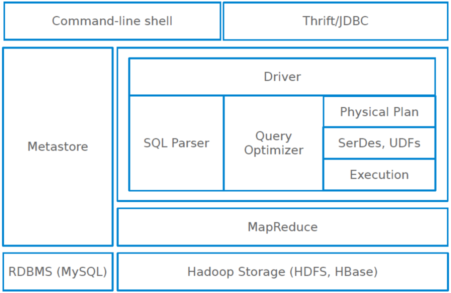
Hive Components (7/8)

- SerDe
  - Serializer/Deserializer allows Hive to read and write table rows in any custom format.



Hive Components (8/8)

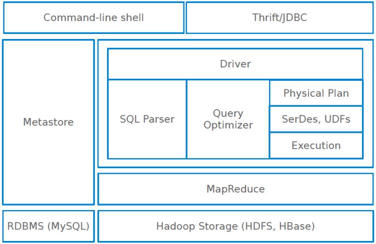
- Metastore
  - The system catalog.
  - Contains metadata about the tables.
  - Metadata is specified during table creation and reused every time the table is referenced in HiveQL.
  - Metadatas are stored on either a traditional relational database, e.g., MySQL, or file system and not HDFS.

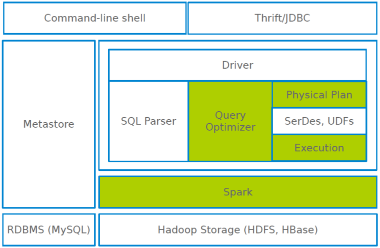


Spark SQL

Shark

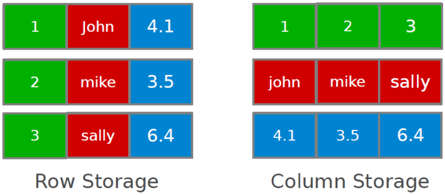
▶ Shark modified the Hive backend to run over Spark.





In-Memory Column Store

- Simply caching Hive records as JVM objects is inefficient.
- 12 to 16 bytes of overhead per object in JVM implementation:
  - e.g., storing a 270MB table as JVM objects uses approximately 971 MB of memory.
- Shark employs column-oriented storage using arrays of primitive objects.



Shark Limitations

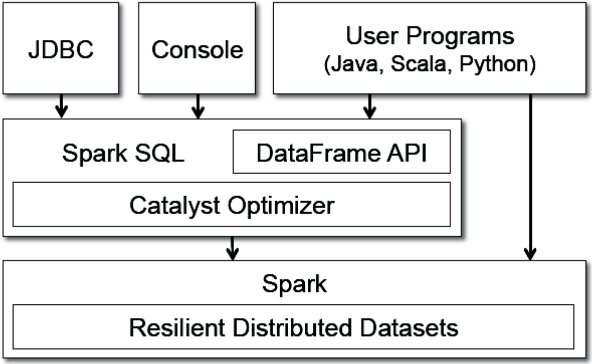
- Limited integration with Spark programs.
- Hive optimizer not designed for Spark.

From Shark to Spark SQL

- Borrows from Shark
  - Hive data loading
  - In-memory column store

- Adds by Spark
  - RDD-aware optimizer (catalyst optimizer)
  - Adds schema to RDD (DataFrame)
  - Rich language interfaces

Spark and Spark SQL

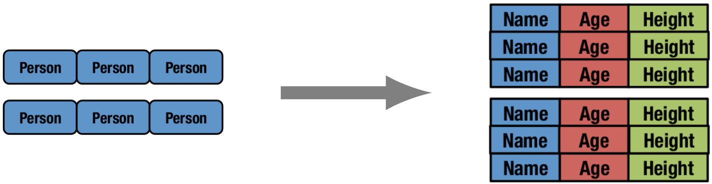


DataFrame

- A DataFrame is a distributed collection of rows with a homogeneous schema.
- it is equivalent to a table in a relational database.
- It can also be manipulated in similar ways to RDDs.
- DataFrames are lazy.

Adding Schema to RDDs

- Spark + RDD: functional transformations on partitioned collections of opaque objects.
- SQL + DataFrame: declarative transformations on partitioned collections of tuples.



Creating DataFrames

- The entry point into all functionality in Spark SQL is the `SQLContext`.
- With a `SQLContext`, applications can create DataFrames from an existing RDD, from a Hive table, or from data sources.

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json(...)
```

DataFrame Operations (1/2)

- Domain-specific language for structured data manipulation.

```
// Show the content of the DataFrame
df.show()
// age  name
// null Michael
// 30   Andy
// 19   Justin

// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// name
// Michael
// Andy
// Justin
```

DataFrame Operations (2/2)

- Domain-specific language for structured data manipulation.

```
// Select everybody, but increment the age by 1
df.select(df("name"), df("age") + 1).show()
// name    (age + 1)
// Michael null
// Andy    31
// Justin  20

// Select people older than 21
df.filter(df("age") > 21).show()
// age name
// 30  Andy

// Count people by age
df.groupBy("age").count().show()
// age  count
// null 1
// 19   1
// 30   1
```

Running SQL Queries Programmatically

- Running SQL queries programmatically and returns the result as a DataFrame.
- Using the `sql` function on a `SQLContext`.

```
val sqlContext = ...  // An existing SQLContext
val df = sqlContext.sql("SELECT * FROM table")
```

Converting RDDs into DataFrames

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

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(...).map(_.split(","))
               .map(p => Person(p(0), p(1).trim.toInt)).toDF()
people.registerTempTable("people")
```

Converting RDDs into DataFrames

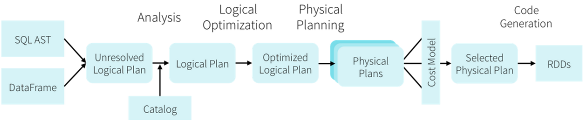
```
// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(...).map(_.split(","))
               .map(p => Person(p(0), p(1).trim.toInt)).toDF()
people.registerTempTable("people")
```

```
// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext
    .sql("SELECT name, age FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are DataFrames.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
teenagers.map(t => "Name: " + t.getAs[String]("name")).collect()
         .foreach(println)
```

Catalyst Optimizer

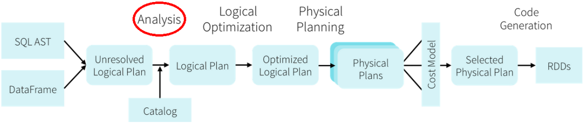
Using Catalyst in Spark SQL

- The Catalyst is used in four phases:
  1. Analyzing a logical plan to resolve references
  2. Logical plan optimization
  3. Physical planning
  4. Code generation to compile parts of the query to Java bytecode

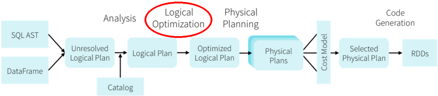


Query Planning in Spark SQL - Analysis

- A relation may contain unresolved attribute references or relations
- Example:
  - `SQL query SELECT col FROM sales`
  - The `col` is unresolved until we look up the table `sales`.
- Spark SQL uses Catalyst rules and a Catalog object that tracks the tables in all data sources to resolve these attributes.

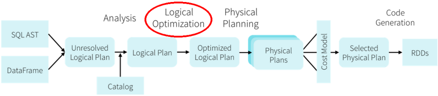


Query Planning in Spark SQL - Logical Optimization (1/5)



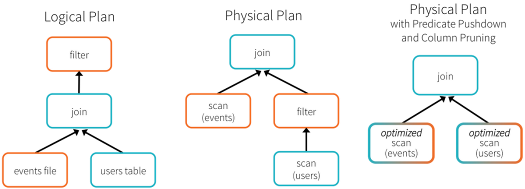
- Applies standard rule-based optimizations to the logical plan.

Query Planning in Spark SQL - Logical Optimization (1/5)



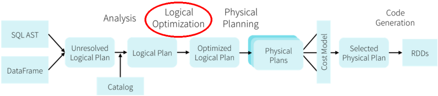
- Applies standard rule-based optimizations to the logical plan.

```
val users = sqlContext.read.parquet("...")
val events = sqlContext.read.parquet("...")
val joined = events.join(users, ...)
val result = joined.select(...)
```



Query Planning in Spark SQL - Logical Optimization (2/5)

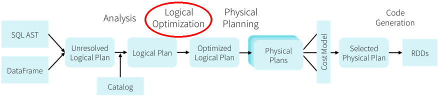
- Null propagation and constant folding
  - Replace expressions that can be evaluated with some literal value to the value.
  - `1 + null` $\Rightarrow$ `null`
  - `1 + 2` $\Rightarrow$ `3`



Query Planning in Spark SQL - Logical Optimization (2/5)

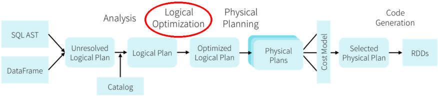
- Null propagation and constant folding
  - Replace expressions that can be evaluated with some literal value to the value.
  - `1 + null` ⇒ `null`
  - `1 + 2` ⇒ `3`

- Boolean simplification
  - Simplifies boolean expressions that can be determined.
  - `false AND x` ⇒ `false`
  - `true AND x` ⇒ `x`
  - `true OR x` ⇒ `true`
  - `false OR x` ⇒ `x`



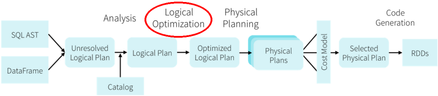
Query Planning in Spark SQL - Logical Optimization (3/5)

- Simplify filters
  - Removes filters that can be evaluated trivially.
  - `Filter(true, child)` ⇒ `child`
  - `Filter(false, child)` ⇒ `empty`

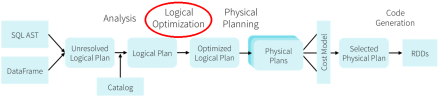


Query Planning in Spark SQL - Logical Optimization (3/5)

- **Simplify filters**
  - Removes filters that can be evaluated trivially.
  - `Filter(true, child)` ⇒ `child`
  - `Filter(false, child)` ⇒ `empty`
- **Combine filters**
  - Merges two filters.
  - `Filter($fc, Filter($nc, child))`
    ⇒
    `Filter(AND($fc, $nc), child)`

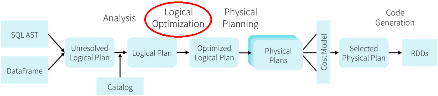


- Push predicate through project
  - Pushes filter operators through project operator.
  - `Filter(i == 1, Project(i, j, child))`
    $\Rightarrow$
    `Project(i, j, Filter(i == 1, child))`

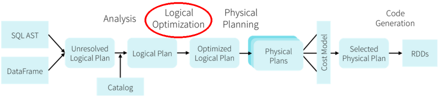


Query Planning in Spark SQL - Logical Optimization (4/5)

- **Push predicate through project**
  - Pushes filter operators through project operator.
  - `Filter(i == 1, Project(i, j, child))`
    $\Rightarrow$
    `Project(i, j, Filter(i == 1, child))`
- **Push predicate through join**
  - Pushes filter operators through join operator.
  - `Filter("left.i".attr == 1, Join(left, right))`
    $\Rightarrow$
    `Join(Filter(i == 1, left), right)`



- Column pruning
  - Eliminates the reading of unused columns.
  - `Join(left, right, LeftSemi, "left.id".attr == "right.id".attr)`
    $\Rightarrow$
    `Join(left, Project(id, right), LeftSemi)`

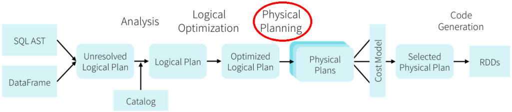


Query Planning in Spark SQL - Physical Optimization

- Generates one or more physical plans using physical operators that match the Spark execution engine.

- Selects a plan using a cost model: based on join algorithms.
  - Broadcast join for small relations

- Performs rule-based physical optimizations
  - Pipelining projections or filters into one map operation.
  - Pushing operations from the logical plan into data sources that support predicate or projection pushdown.



Project Tungsten

Project Tungsten

- Spark workloads are increasingly bottlenecked by CPU and memory use rather than IO and network communication.

- Goals of Project Tungsten: improve the memory and CPU efficiency of Spark backend execution and push performance closer to the limits of modern hardware.

Project Tungsten Initiatives

- Perform manual memory management instead of relying on Java objects.
  - Reduce memory footprint.
  - Eliminate garbage collection overheads.
  - Use java.unsafe and off heap memory.
- Code generation for expression evaluation.
  - Reduce virtual function calls and interpretation overhead.
- Cache conscious sorting.
  - Reduce bad memory access patterns.

Summary

Summary

- RDD: a distributed memory abstraction
- Two types of operations: transformations and actions
- Lineage graph
- DataFrame: structured processing
- Logical and physical plans
- Catalyst optmizer
- Tungsten project

Questions?