



# Structured Data Processing - Spark SQL

Amir H. Payberah  
payberah@kth.se  
2021-09-20



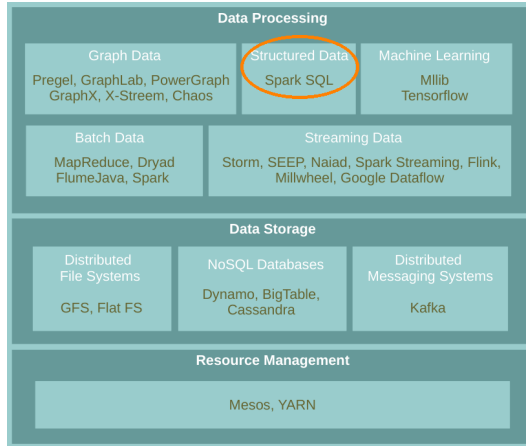


## The Course Web Page

`https://id2221kth.github.io`

`https://tinyurl.com/f6x544h`

# Where Are We?



# Motivation

Structured Data



Unstructured Data





# Hive

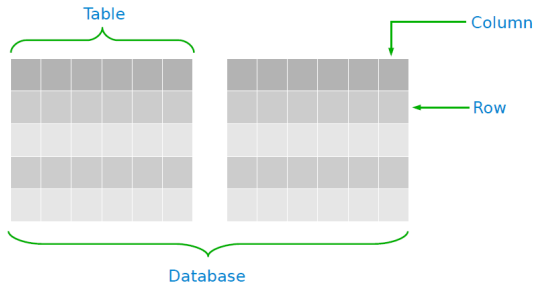
- ▶ A system for **managing** and **querying structured data** built on top of **MapReduce**.
- ▶ Converts a query to a **series of MapReduce phases**.
- ▶ Initially developed by **Facebook**.



# 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
- ▶ **Data Definition Language (DDL)** operations
  - Create, Alter, Drop

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





## Hive API (2/2)

- ▶ Data Manipulation Language (DML) operations
  - Load and Insert (overwrite)
  - Does **not** support **updating** and **deleting**

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



## Hive API (2/2)

- ▶ Data Manipulation Language (DML) operations
  - Load and Insert (overwrite)
  - Does **not** support **updating** and **deleting**

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

- ▶ Query operations
  - Select, Filter, Join, Groupby

```
-- 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.



## 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.



## 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.

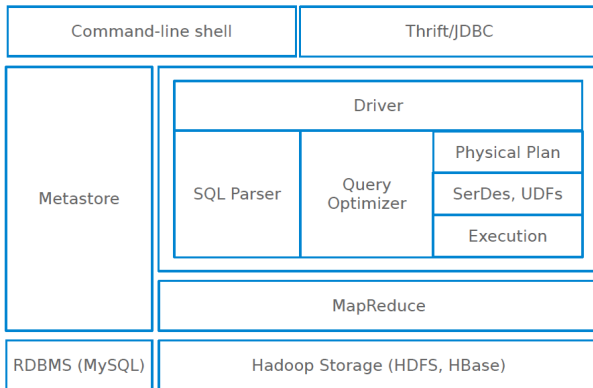


## Executing SQL Questions

- ▶ 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 tasks**.



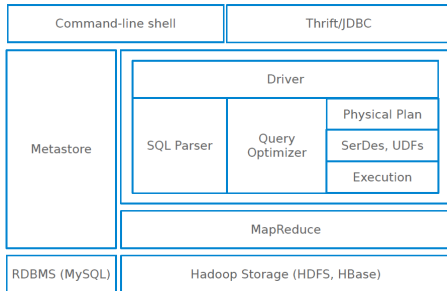
# Hive Architecture





# Hive Architecture - Driver

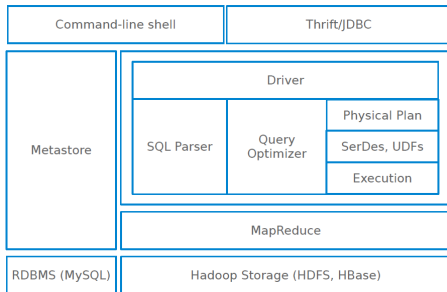
- ▶ Manages the **life cycle** of a HiveQL statement during compilation, optimization and execution.





# Hive Architecture - Compiler (Parser/Query Optimizer)

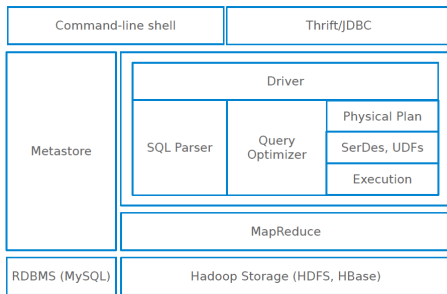
- ▶ Translates the HiveQL statement into a **logical plan** and **optimizes it**.





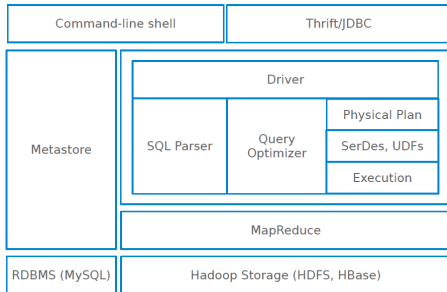
# Hive Architecture - Physical Plan

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



# Hive Architecture - Execution Engine

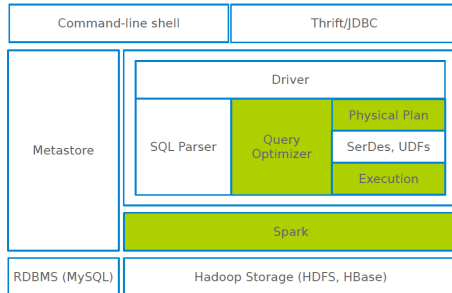
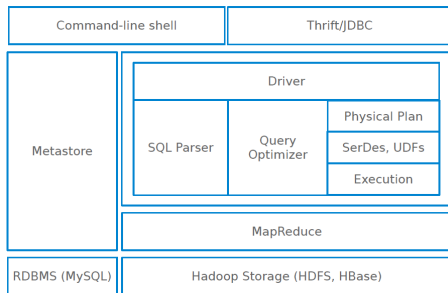
- ▶ The driver submits the individual mapreduce jobs from the DAG to the execution engine in a **topological order**.





# Spark SQL

- ▶ **Shark** modified the **Hive** backend to run over **Spark**.





# Shark and Hive In-Memory Store

- ▶ Caching Hive records as **JVM objects** is **inefficient**.
  - 12 to 16 bytes of overhead per object in JVM implementation:
- ▶ Shark employs **column-oriented** storage using arrays of **primitive objects**.

|   |       |     |
|---|-------|-----|
| 1 | John  | 4.1 |
| 2 | mike  | 3.5 |
| 3 | sally | 6.4 |

Row Storage

|      |      |       |
|------|------|-------|
| 1    | 2    | 3     |
| john | mike | sally |
| 4.1  | 3.5  | 6.4   |

Column Storage



## Shark Limitations

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



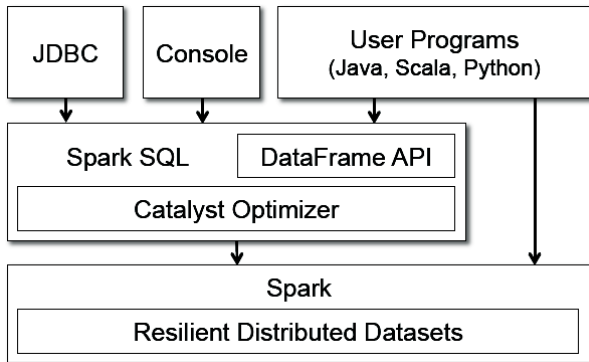
# From Shark to Spark SQL

- ▶ **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



## Structured Data vs. RDD (1/2)

► `case class Account(name: String, balance: Double, risk: Boolean)`



## Structured Data vs. RDD (1/2)

- ▶ `case class Account(name: String, balance: Double, risk: Boolean)`
- ▶ `RDD[Account]`



## Structured Data vs. RDD (1/2)

- ▶ `case class Account(name: String, balance: Double, risk: Boolean)`
- ▶ `RDD[Account]`
- ▶ **RDDs don't know** anything about the **schema** of the data it's dealing with.





## Structured Data vs. RDD (2/2)

- ▶ `case class Account(name: String, balance: Double, risk: Boolean)`
- ▶ `RDD[Account]`
- ▶ A **database/Hive** sees it as a columns of named and typed values.

|                           |                              |                            |
|---------------------------|------------------------------|----------------------------|
| <code>name: String</code> | <code>balance: Double</code> | <code>risk: Boolean</code> |
| <code>name: String</code> | <code>balance: Double</code> | <code>risk: Boolean</code> |
| <code>name: String</code> | <code>balance: Double</code> | <code>risk: Boolean</code> |
| <code>name: String</code> | <code>balance: Double</code> | <code>risk: Boolean</code> |



# DataFrames and DataSets

- ▶ Spark has **two** notions of **structured collections**:
  - **DataFrames**
  - **Datasets**
- ▶ They are **distributed table-like collections** with **well-defined rows and columns**.



# DataFrames and DataSets

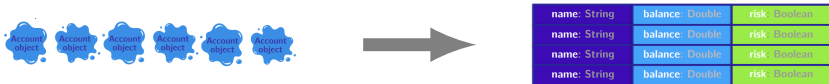
- ▶ Spark has **two** notions of **structured collections**:
  - **DataFrames**
  - **Datasets**
- ▶ They are **distributed table-like collections** with **well-defined rows and columns**.
- ▶ They represent **immutable lazily** evaluated plans.
- ▶ When an **action** is performed on them, Spark performs the **actual transformations** and return the result.

# DataFrame



# DataFrame

- ▶ Consists of a **series of rows** and a **number of columns**.
- ▶ Equivalent to a **table** in a relational database.
- ▶ **Spark + RDD**: **functional** transformations on partitioned collections of **objects**.
- ▶ **SQL + DataFrame**: **declarative** transformations on partitioned collections of **tuples**.





# Schema

- ▶ Defines the **column names and types** of a DataFrame.
- ▶ Assume `people.json` file as an input:

```
{ "name": "Michael", "age": 15, "id": 12 }  
{ "name": "Andy", "age": 30, "id": 15 }  
{ "name": "Justin", "age": 19, "id": 20 }  
{ "name": "Andy", "age": 12, "id": 15 }  
{ "name": "Jim", "age": 19, "id": 20 }  
{ "name": "Andy", "age": 12, "id": 10 }
```



# Schema

- ▶ Defines the **column names and types** of a DataFrame.
- ▶ Assume `people.json` file as an input:

```
{"name": "Michael", "age": 15, "id": 12}  
{"name": "Andy", "age": 30, "id": 15}  
{"name": "Justin", "age": 19, "id": 20}  
{"name": "Andy", "age": 12, "id": 15}  
{"name": "Jim", "age": 19, "id": 20}  
{"name": "Andy", "age": 12, "id": 10}
```

```
val people = spark.read.format("json").load("people.json")  
people.schema
```

*// returns:*

```
StructType(StructField(age, LongType, true),  
StructField(id, LongType, true),  
StructField(name, StringType, true))
```



## Column (1/2)

- ▶ They are like **columns in a table**.
- ▶ `col` returns a reference to a column.
- ▶ `expr` performs transformations on a column.
- ▶ `columns` returns all columns on a DataFrame

```
val people = spark.read.format("json").load("people.json")  
  
col("age")  
  
exp("age + 5 < 32")  
  
people.columns  
// returns: Array[String] = Array(age, id, name)
```



## Column (2/2)

- ▶ Different ways to refer to a column.

```
val people = spark.read.format("json").load("people.json")  
  
people.col("name")  
  
col("name")  
  
column("name")  
  
'name  
  
$"name"  
  
expr("name")
```



## Row

- ▶ A **row** is a **record of data**.
- ▶ They are of type **Row**.
- ▶ Rows do **not have schemas**.

```
import org.apache.spark.sql.Row  
  
val myRow = Row("Seif", 65, 0)
```



## Row

- ▶ A **row** is a **record of data**.
- ▶ They are of type **Row**.
- ▶ Rows do **not have schemas**.
  - The **order of values** should be **the same order as the schema** of the DataFrame to which they might be appended.
- ▶ To access data in rows, you need to specify the **position** that you would like.

```
import org.apache.spark.sql.Row
```

```
val myRow = Row("Seif", 65, 0)
```

```
myRow(0) // type Any  
myRow(0).asInstanceOf[String] // String  
myRow.getString(0) // String  
myRow.getInt(1) // Int
```



## Creating a DataFrame

- ▶ **Two** ways to create a DataFrame:
  1. From an **RDD**
  2. From **raw data sources**





## Creating a DataFrame - From an RDD

- ▶ The schema *automatically* inferred.



## Creating a DataFrame - From an RDD

- ▶ The schema **automatically inferred**.
- ▶ You can use **toDF** to **convert an RDD to DataFrame**.

```
val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1))  
val tupleDF = tupleRDD.toDF("name", "age", "id")
```



## Creating a DataFrame - From an RDD

- ▶ The schema automatically inferred.
- ▶ You can use `toDF` to convert an RDD to DataFrame.

```
val tupleRDD = sc.parallelize(Array(("seif", 65, 0), ("amir", 40, 1)))  
val tupleDF = tupleRDD.toDF("name", "age", "id")
```

- ▶ If RDD contains `case` class instances, Spark infers the attributes from it.

```
case class Person(name: String, age: Int, id: Int)  
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))  
val peopleDF = peopleRDD.toDF
```



## Creating a DataFrame - From Data Source

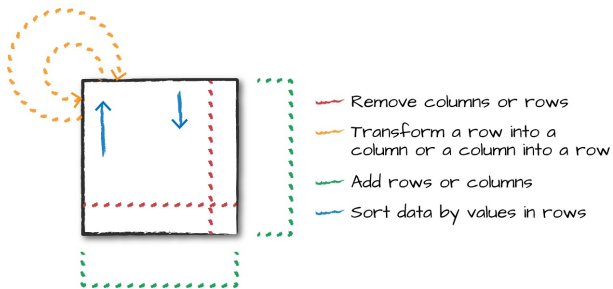
- ▶ Data sources supported by Spark.
  - CSV, JSON, Parquet, ORC, JDBC/ODBC connections, Plain-text files
  - Cassandra, HBase, MongoDB, AWS Redshift, XML, etc.

```
val peopleJson = spark.read.format("json").load("people.json")

val peopleCsv = spark.read.format("csv")
  .option("sep", ";")
  .option("inferSchema", "true")
  .option("header", "true")
  .load("people.csv")
```

# DataFrame Transformations (1/4)

- ▶ Add and remove rows or columns
- ▶ Transform a row into a column (or vice versa)
- ▶ Change the order of rows based on the values in columns



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]



## DataFrame Transformations (2/4)

- ▶ `select` and `selectExpr` allow to do the **DataFrame equivalent** of **SQL queries** on a table of data.

```
// select  
people.select("name", "age", "id").show(2)  
people.select(col("name"), expr("age + 3")).show()
```



## DataFrame Transformations (2/4)

- ▶ `select` and `selectExpr` allow to do the **DataFrame equivalent** of SQL queries on a table of data.

```
// select
people.select("name", "age", "id").show(2)
people.select(col("name"), expr("age + 3")).show()
```

```
// selectExpr
people.selectExpr("*", "(age < 20) as teenager").show()
people.selectExpr("avg(age)", "count(distinct(name))", "sum(id)").show()
```



## DataFrame Transformations (3/4)

- ▶ `filter` and `where` both `filter` rows.
- ▶ `distinct` can be used to extract unique rows.

```
people.filter(col("age") < 20).show()
```

```
people.where("age < 20").show()
```

```
people.select("name").distinct().show()
```





## DataFrame Transformations (4/4)

- ▶ `withColumn` adds a new column to a DataFrame.
- ▶ `withColumnRenamed` renames a column.
- ▶ `drop` removes a column.

```
// withColumn
people.withColumn("teenager", expr("age < 20")).show()

// withColumnRenamed
people.withColumnRenamed("name", "username").columns

// drop
people.drop("name").columns
```



## DataFrame Actions

- ▶ Like RDDs, DataFrames also have their own set of actions.
- ▶ `collect`: returns an `array` that contains all of `rows` in this DataFrame.
- ▶ `count`: returns the `number of rows` in this DataFrame.
- ▶ `first` and `head`: returns the `first row` of the DataFrame.
- ▶ `show`: displays the `top 20 rows` of the DataFrame in a tabular form.
- ▶ `take`: returns the `first n rows` of the DataFrame.

# Aggregation



# Aggregation

- ▶ In an **aggregation** you specify
  - A **key or grouping**
  - An **aggregation function**
- ▶ The given function must produce **one** result for **each group**.



## Grouping Types

- ▶ Summarizing a complete DataFrame
- ▶ Group by
- ▶ Windowing



## Grouping Types

- ▶ Summarizing a complete DataFrame
- ▶ Group by
- ▶ Windowing



## Summarizing a Complete DataFrame Functions (1/2)

- ▶ `count` returns the **total number of values**.
- ▶ `countDistinct` returns the **number of unique groups**.
- ▶ `first` and `last` return the **first and last value** of a DataFrame.

```
val people = spark.read.format("json").load("people.json")  
people.select(count("age")).show()  
people.select(countDistinct("name")).show()  
people.select(first("name"), last("age")).show()
```



## Summarizing a Complete DataFrame Functions (2/2)

- ▶ `min` and `max` extract the **minimum and maximum values** from a DataFrame.
- ▶ `sum` **adds all the values** in a column.
- ▶ `avg` calculates the **average**.

```
val people = spark.read.format("json").load("people.json")  
people.select(min("name"), max("age"), max("id")).show()  
people.select(sum("age")).show()  
people.select(avg("age")).show()
```





## Grouping Types

- ▶ Summarizing a complete DataFrame
- ▶ Group by
- ▶ Windowing



## Group By (1/3)

- ▶ Perform aggregations on **groups** in the data.
- ▶ Typically on **categorical data**.
- ▶ We do this grouping in **two phases**:
  1. **Specify the column(s)** on which we would like to group.
  2. Specify the **aggregation(s)**.



## Group By (2/3)

▶ Grouping with **expressions**

- Rather than passing that function as an expression into a **select** statement, we **specify it as within agg**.

```
val people = spark.read.format("json").load("people.json")
people.groupBy("name").agg(count("age").alias("ageagg")).show()
```



## Group By (3/3)

### ▶ Grouping with Maps

- Specify transformations as a **series of Maps**
- The **key** is the **column**, and the **value** is the **aggregation function** (as a string).

```
val people = spark.read.format("json").load("people.json")
people.groupBy("name").agg("age" -> "count", "age" -> "avg", "id" -> "max").show()
```

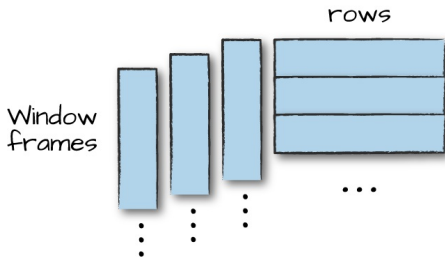


## Grouping Types

- ▶ Summarizing a complete DataFrame
- ▶ Group by
- ▶ **Windowing**

## Windowing (1/2)

- ▶ Computing some aggregation on a specific **window** of data.
- ▶ The **window** determines **which rows** will be passed in to this function.
- ▶ You define them by using a **reference to the current data**.
- ▶ A **group of rows** is called a **frame**.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]



## Windowing (2/2)

- ▶ Unlike grouping, here **each row** can fall into **one or more frames**.

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions.col

val people = spark.read.format("json").load("people.json")

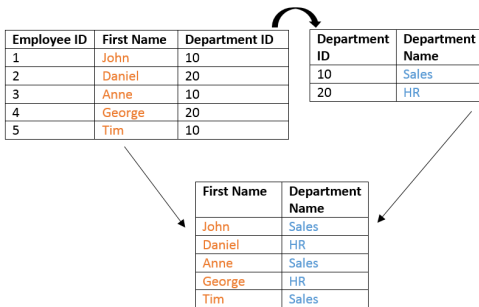
val windowSpec = Window.rowsBetween(-1, 1)
val avgAge = avg(col("age")).over(windowSpec)
people.select(col("name"), col("age"), avgAge.alias("avg_age")).show
```

# Joins



# Joins

- ▶ Joins are **relational** constructs you use to **combine relations together**.
- ▶ Different **join types**: inner join, outer join, left outer join, right outer join, left semi join, left anti join, cross join





## Joins Example

```
val person = Seq((0, "Seif", 0), (1, "Amir", 1), (2, "Sarunas", 1))
                .toDF("id", "name", "group_id")

val group = Seq((0, "SICS/KTH"), (1, "KTH"), (2, "SICS"))
              .toDF("id", "department")
```



## Joins Example - Inner

```
val joinExpression = person.col("group_id") === group.col("id")  
  
var joinType = "inner"  
  
person.join(group, joinExpression, joinType).show()
```

```
+---+-----+-----+---+-----+  
| id|  name|group_id| id|department|  
+---+-----+-----+---+-----+  
|  0|   Seif|      0|  0|  SICS/KTH|  
|  1|   Amir|      1|  1|      KTH|  
|  2|Sarunas|      1|  1|      KTH|  
+---+-----+-----+---+-----+
```



## Joins Example - Outer

```
val joinExpression = person.col("group_id") === group.col("id")  
  
var joinType = "outer"  
  
person.join(group, joinExpression, joinType).show()
```

```
+-----+-----+-----+-----+  
| id| name|group_id| id|department|  
+-----+-----+-----+-----+  
| 1| Amir| 1| 1| KTH|  
| 2| Sarunas| 1| 1| KTH|  
| null| null| null| 2| SICS|  
| 0| Seif| 0| 0| SICS/KTH|  
+-----+-----+-----+-----+
```

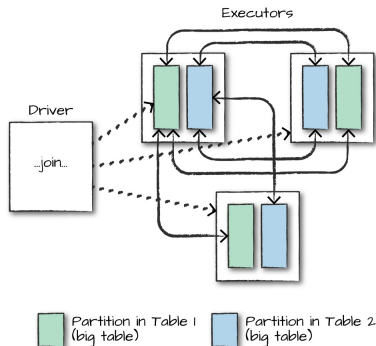


## Joins Communication Strategies

- ▶ Two different communication ways during joins:
  - Shuffle join: big table to big table
  - Broadcast join: big table to small table

# Shuffle Join

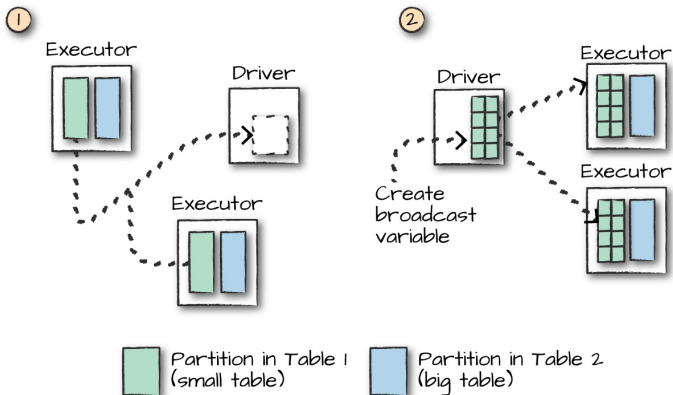
- ▶ Every node talks to **every other node**.
- ▶ They share data according to **which node** has a **certain key or set of keys**.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

# Broadcast Join

- ▶ When the table is **small enough** to fit into the memory of a single worker node.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

# SQL





# SQL

- ▶ You can run **SQL queries** on views/tables via the method `sql` on the `SparkSession` object.

```
spark.sql("SELECT * from people_view").show()
```

```
+---+---+-----+
|age| id|  name|
+---+---+-----+
| 15| 12|Michael|
| 30| 15|  Andy|
| 19| 20| Justin|
| 12| 15|  Andy|
| 19| 20|   Jim|
| 12| 10|  Andy|
+---+---+-----+
```



## Temporary View

- ▶ `createOrReplaceTempView` creates (or replaces) a lazily evaluated `view`.
- ▶ You can use it like a `table` in Spark SQL.

```
people.createOrReplaceTempView("people_view")  
  
val teenagersDF = spark.sql("SELECT name, age FROM people_view WHERE age BETWEEN 13 AND 19")
```

# DataSet



## Untyped API with DataFrame

- ▶ DataFrames elements are Rows, which are generic untyped JVM objects.
- ▶ Scala compiler cannot type check Spark SQL schemas in DataFrames.



## Untyped API with DataFrame

- ▶ DataFrames elements are **Rows**, which are **generic untyped JVM objects**.
- ▶ Scala compiler **cannot type check** Spark SQL **schemas** in DataFrames.
- ▶ The following code **compiles**, but you get a **runtime exception**.
  - `id_num` is not in the DataFrame columns `[name, age, id]`

```
// people columns: ("name", "age", "id")  
val people = spark.read.format("json").load("people.json")  
  
people.filter("id_num < 20") // runtime exception
```



## Why DataSet?

- ▶ Assume the following example

```
case class Person(name: String, age: BigInt, id: BigInt)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF
```



## Why DataSet?

- ▶ Assume the following example

```
case class Person(name: String, age: BigInt, id: BigInt)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF
```

- ▶ Now, let's use `collect` to bring back it to the master.

```
val collectedPeople = peopleDF.collect()
// collectedPeople: Array[org.apache.spark.sql.Row]
```



## Why DataSet?

- ▶ Assume the following example

```
case class Person(name: String, age: BigInt, id: BigInt)
val peopleRDD = sc.parallelize(Array(Person("seif", 65, 0), Person("amir", 40, 1)))
val peopleDF = peopleRDD.toDF
```

- ▶ Now, let's use `collect` to bring back it to the master.

```
val collectedPeople = peopleDF.collect()
// collectedPeople: Array[org.apache.spark.sql.Row]
```

- ▶ What is in `Row`?





## Why DataSet?

- ▶ To be able to work with the collected values, we should **cast** the **Rows**.
  - How many **columns**?
  - What **types**?

```
// Person(name: Sting, age: BigInt, id: BigInt)

val collectedList = collectedPeople.map {
  row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int], row(2).asInstanceOf[Int])
}
```



## Why DataSet?

- ▶ To be able to work with the collected values, we should **cast** the **Rows**.
  - How many **columns**?
  - What **types**?

```
// Person(name: Sting, age: BigInt, id: BigInt)

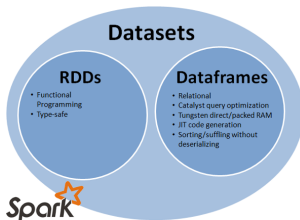
val collectedList = collectedPeople.map {
  row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int], row(2).asInstanceOf[Int])
}
```

- ▶ But, what if we cast the **types wrong**?
- ▶ Wouldn't it be nice if we could have both **Spark SQL optimizations** and **typesafety**?

# DataSet

- ▶ **Datasets** can be thought of as **typed** distributed collections of data.
- ▶ **Dataset** API unifies the **DataFrame** and **RDD** APIs.
- ▶ You can consider a **DataFrame** as an alias for **Dataset [Row]**, where a **Row** is a **generic untyped JVM object**.

```
type DataFrame = Dataset[Row]
```



[<http://why-not-learn-something.blogspot.com/2016/07/apache-spark-rdd-vs-dataframe-vs-dataset.html>]



# Structured APIs in Spark

Structured APIs In Spark

|                 | SQL     | DataFrames   | Datasets     |
|-----------------|---------|--------------|--------------|
| Syntax Errors   | Runtime | Compile Time | Compile Time |
| Analysis Errors | Runtime | Runtime      | Compile Time |

[J.S. Damji et al., Learning Spark - Lightning-Fast Data Analytics]



## Creating DataSets

- ▶ To convert a **sequence** or an **RDD** to a **Dataset**, we can use `toDS()`.
- ▶ You can call `as[SomeCaseClass]` to convert the **DataFrame** to a Dataset.

```
case class Person(name: String, age: BigInt, id: BigInt)
val personSeq = Seq(Person("Max", 33, 0), Person("Adam", 32, 1))
```



## Creating DataSets

- ▶ To convert a **sequence** or an **RDD** to a **Dataset**, we can use `toDS()`.
- ▶ You can call `as[SomeCaseClass]` to convert the **DataFrame** to a Dataset.

```
case class Person(name: String, age: BigInt, id: BigInt)
val personSeq = Seq(Person("Max", 33, 0), Person("Adam", 32, 1))
```

```
val ds1 = sc.parallelize(personSeq).toDS
```



## Creating DataSets

- ▶ To convert a **sequence** or an **RDD** to a **Dataset**, we can use `toDS()`.
- ▶ You can call `as[SomeCaseClass]` to convert the **DataFrame** to a Dataset.

```
case class Person(name: String, age: BigInt, id: BigInt)
val personSeq = Seq(Person("Max", 33, 0), Person("Adam", 32, 1))
```

```
val ds1 = sc.parallelize(personSeq).toDS
```

```
val ds2 = spark.read.format("json").load("people.json").as[Person]
```



# DataSet Transformations

- ▶ Transformations on **Datasets** are the same as those that we had on **DataFrames**.
- ▶ Datasets allow us to specify **more complex and strongly typed** transformations.

```
case class Person(name: String, age: BigInt, id: BigInt)

val people = spark.read.format("json").load("people.json").as[Person]

people.filter(x => x.age < 40).show()

people.map(x => (x.name, x.age + 5, x.id)).show()
```

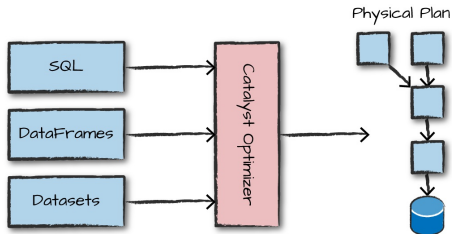




# Structured Data Execution

# Structured Data Execution Steps

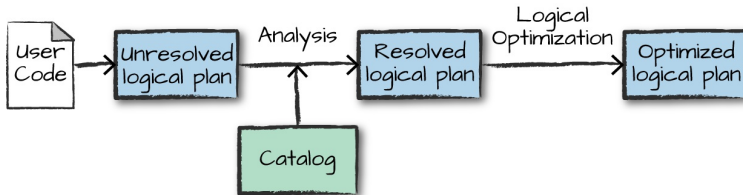
- ▶ 1. Write **DataFrame/Dataset/SQL Code**.
- ▶ 2. If **valid code**, Spark converts this to a **logical plan**.
- ▶ 3. Spark transforms this **logical plan** to a **Physical Plan**
  - Checking for **optimizations** along the way.
- ▶ 4. Spark then executes this **physical plan** (**RDD manipulations**) on the cluster.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

## Logical Planning (1/2)

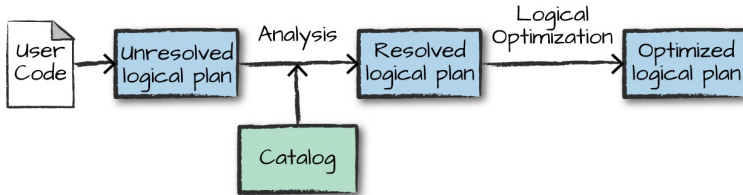
- ▶ The **logical plan** represents a set of **abstract transformations**.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

## Logical Planning (1/2)

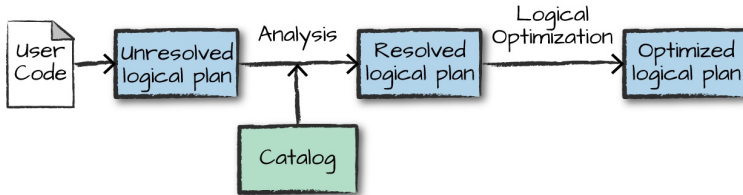
- ▶ The **logical plan** represents a set of **abstract transformations**.
- ▶ This plan is **unresolved**.
  - The **code might be valid**, the **tables/columns** that it refers to **might not exist**.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

## Logical Planning (1/2)

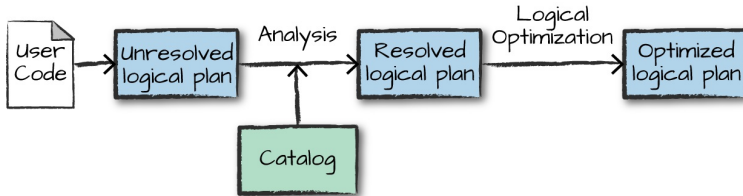
- ▶ The **logical plan** represents a set of **abstract transformations**.
- ▶ This plan is **unresolved**.
  - The **code might be valid**, the **tables/columns** that it refers to **might not exist**.
- ▶ Spark uses the **catalog**, a **repository of all table and DataFrame information**, to resolve columns and tables in the analyzer.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

## Logical Planning (2/2)

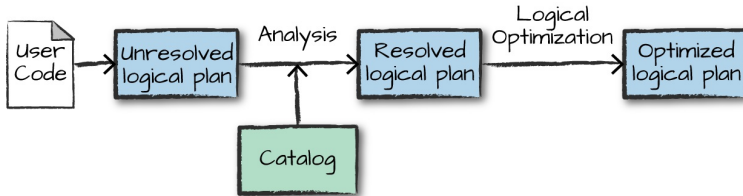
- ▶ The analyzer might **reject** the unresolved logical plan.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

## Logical Planning (2/2)

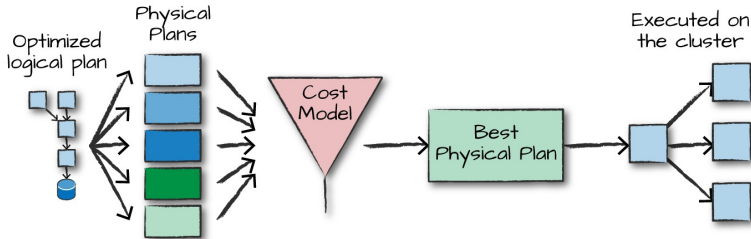
- ▶ The analyzer might **reject** the unresolved logical plan.
- ▶ If the analyzer can resolve it, the result is passed through the **Catalyst optimizer**.
- ▶ It converts the **user's set of expressions** into the most **optimized version**.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

# Physical Planning

- ▶ The **physical plan** specifies **how the logical plan will execute on the cluster**.
- ▶ Physical planning results in a **series of RDDs and transformations**.



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]





## Execution

- ▶ Upon selecting a physical plan, Spark runs all of this code over RDDs.
- ▶ Spark performs further optimizations at runtime.
- ▶ Finally the result is returned to the user.

# Optimization



# Optimization

- ▶ Spark SQL comes with **two specialized backend components**:
  - **Catalyst**: a query optimizer
  - **Tungsten**: off-heap serializer

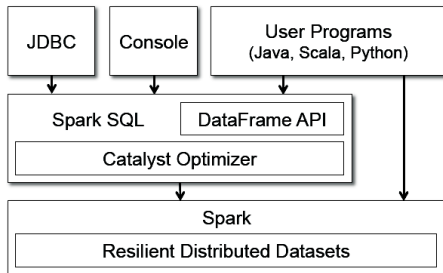


# Catalyst Optimizer



# Catalyst Optimizer

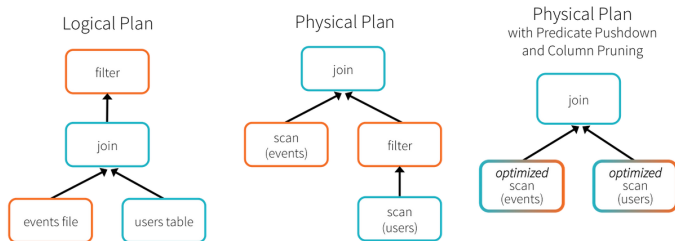
- ▶ **Catalyst** is Spark SQL query optimizer.
- ▶ It compiles Spark SQL queries to **RDDs and transformations**.
- ▶ Optimization includes
  - **Reordering** operations
  - **Reduce** the amount of data we must read
  - **Pruning** unneeded partitioning



# Catalyst Optimizer - Logical Optimization

- ▶ 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(...)
```



# Tungsten



# Tungsten

- ▶ Spark workloads are increasingly **bottlenecked by CPU and memory** use rather than **IO and network communication**.
- ▶ **Tungsten** improves the **memory and CPU efficiency** of Spark backend execution and push performance closer to the limits of modern hardware.
- ▶ It provides
  - **Column-based** datastore
  - **Off-heap** memory management
  - Highly-specialized **data encoders**



# Tungsten - Column-Based

- ▶ Most **table operations** are on specific **columns/attributes** of a dataset.
- ▶ To store data, **group them by column**, instead of row.
- ▶ **Faster lookup** of data associated with specific **column/attribute**.

|   |       |     |
|---|-------|-----|
| 1 | John  | 4.1 |
| 2 | mike  | 3.5 |
| 3 | sally | 6.4 |

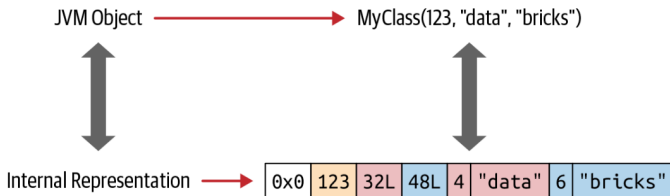
Row Storage

|      |      |       |
|------|------|-------|
| 1    | 2    | 3     |
| john | mike | sally |
| 4.1  | 3.5  | 6.4   |

Column Storage

# Tungsten - Off-Heap

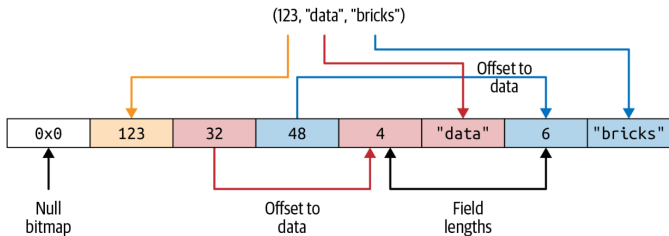
- ▶ Perform **manual memory management** instead of relying on Java objects.
- ▶ Eliminate **garbage collection overheads**.
- ▶ Use `java.unsafe` and **off heap memory**.



[J.S. Damji et al., Learning Spark - Lightning-Fast Data Analytics]

# Tungsten - Data Encoder

- ▶ Tungsten can take **schema information** and **tightly pack serialized data** into memory.
- ▶ **More data** can fit in memory.
- ▶ We have **faster** serialization and deserialization.



[J.S. Damji et al., Learning Spark - Lightning-Fast Data Analytics]

# Summary



## Summary

- ▶ RDD vs. DataFrame vs. DataSet
- ▶ Logical and physical plans
- ▶ Catalyst optimizer
- ▶ Tungsten project



## References

- ▶ M. Zaharia et al., “Spark: The Definitive Guide”, O’Reilly Media, 2018 - Chapters 4-11.
- ▶ M. Armbrust et al., “Spark SQL: Relational data processing in spark”, ACM SIGMOD, 2015.
- ▶ Some slides were derived from Heather Miller’s slides:  
<http://heather.miller.am/teaching/cs4240/spring2018>

Questions?