Big Data Technologies

SQL with Spark

Lionel Fillatre

Polytech Nice Sophia

lionel.fillatre@univ-cotedazur.fr

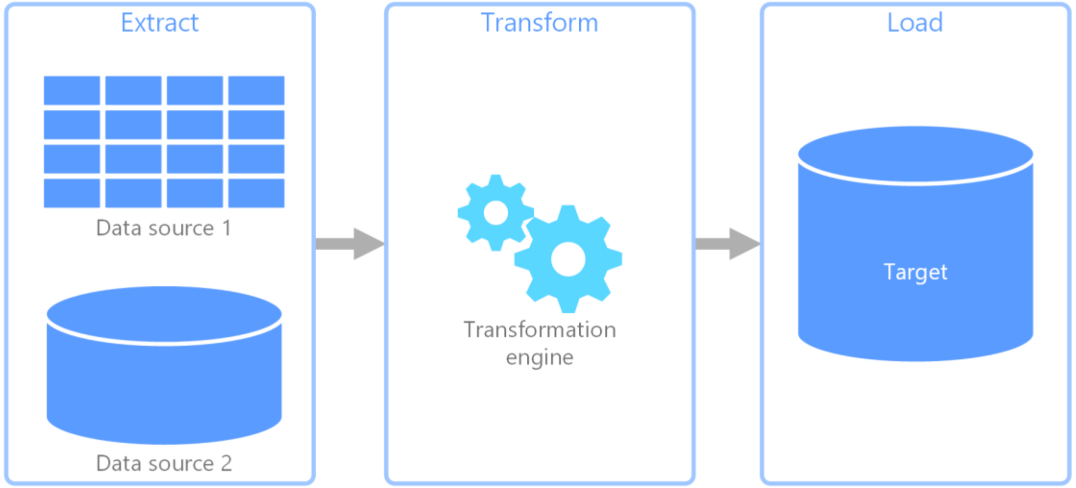
Outlines

- SparkSQL Concepts
- SparkSQL Guide
- Catalyst
- SparkSQL Conclusion

Spark SQL

Challenges and Solutions

- **Challenges**
  - Perform ETL (Extract-Transform-Load) to and from various (semi- or unstructured) data sources
  - Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems.
- **Solutions**
  - A DataFrame API that can perform relational operations on both external data sources and Spark’s built-in RDDs.
  - A highly extensible optimizer, Catalyst, that uses features of Scala to add composable rule, control code generation, and define extensions.



About Spark SQL

- Part of the core distribution since Spark 1.0 (April 2014)
- Spark Engine does not understand the structure of the data in RDDs or the semantics of user functions → limited optimization.
- Runs SQL / HiveQL queries, optionally alongside or replacing existing Hive deployments

```sql
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

Programming Interface

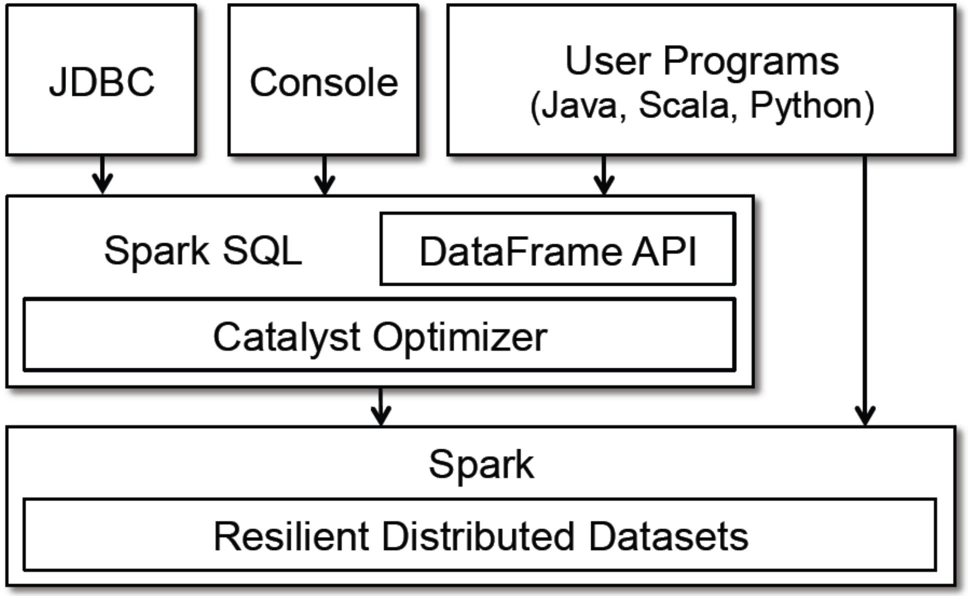


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

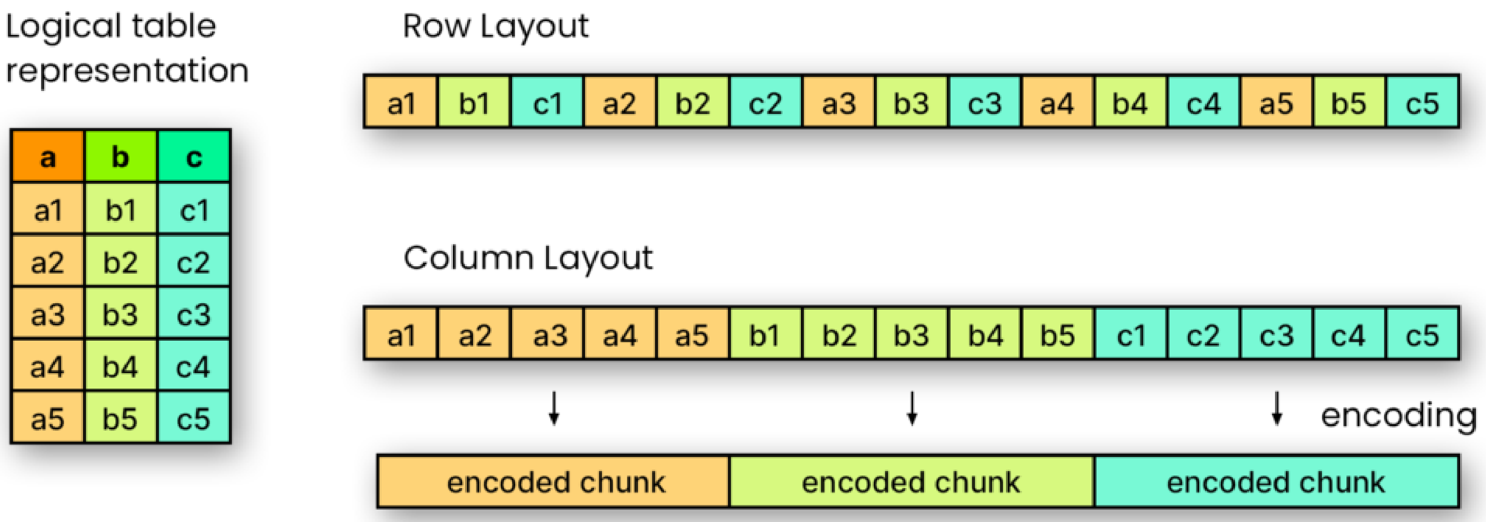
SparkSQL Concepts

Various Data Sources Available in SparkSQL

- **Parquet Files :** It is a columnar format that is supported by many other data processing systems. Parquet files automatically preserves the schema of the original data.
- **ORC Files:** It is a free and open-source column-oriented data storage format (stores data tables by column rather than by row).
- **JSON Files:** It is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value)
- **Hive Tables**
- **JDBC To Other Databases**
- **Avro Files:** Avro is a data serialization system.

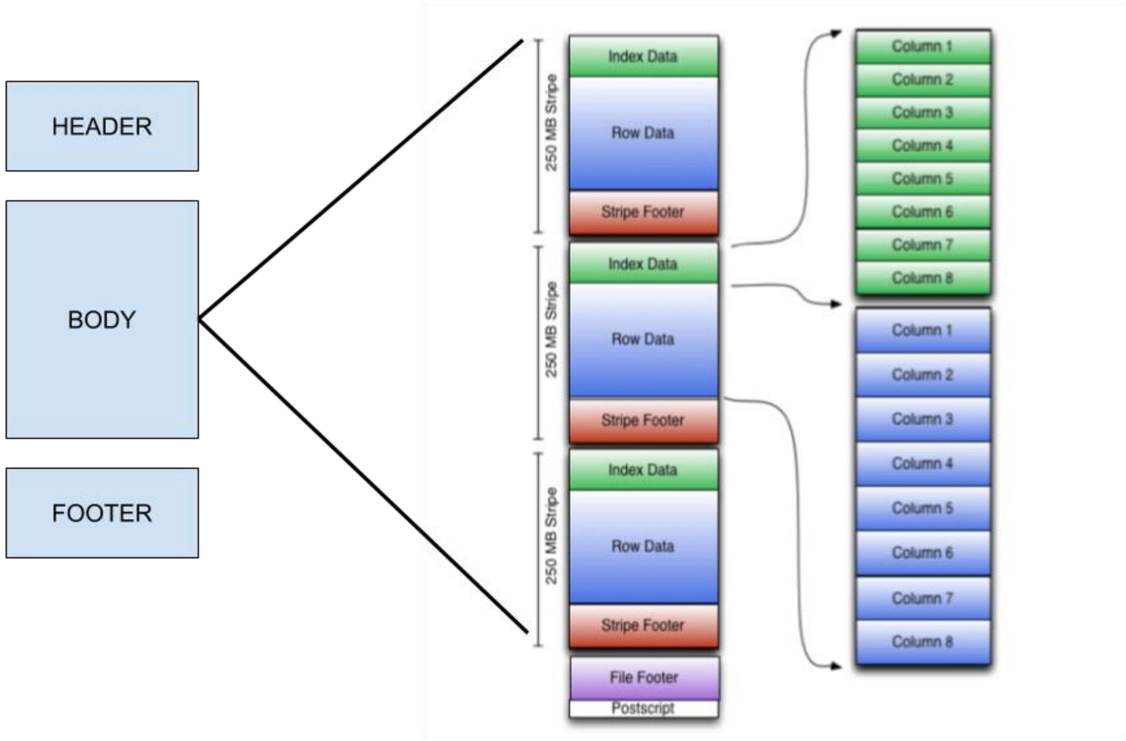
Apache Parquet File Format

- Apache Parquet is an open source file format that stores data in columnar format (as opposed to row format).
- Row-based formats such as CSV and JSON are (mostly) readable by humans, whereas column-based formats are optimized for computers.
- As a columnar file format, Apache Parquet can be read by computers much more efficiently and cost-effectively than other formats.



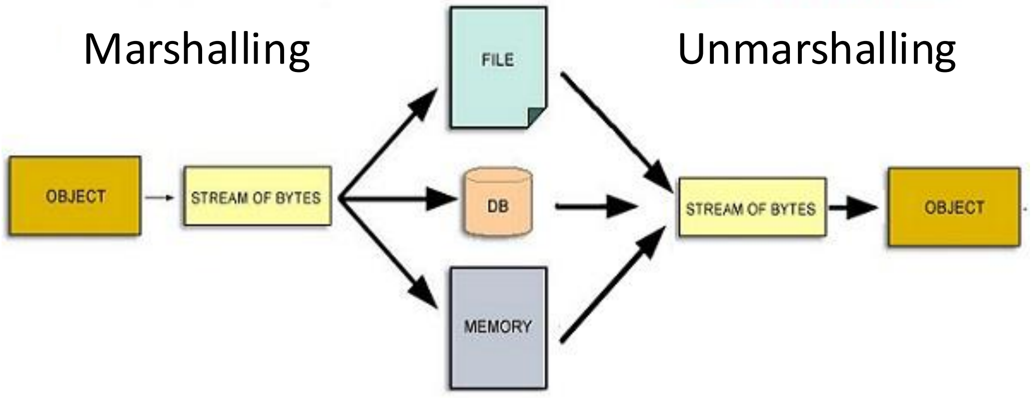
ORC File Format

- The Optimized Row Columnar (ORC) file format provides a highly efficient way to store Hive data.
- An ORC file contains groups of row data called **stripes**

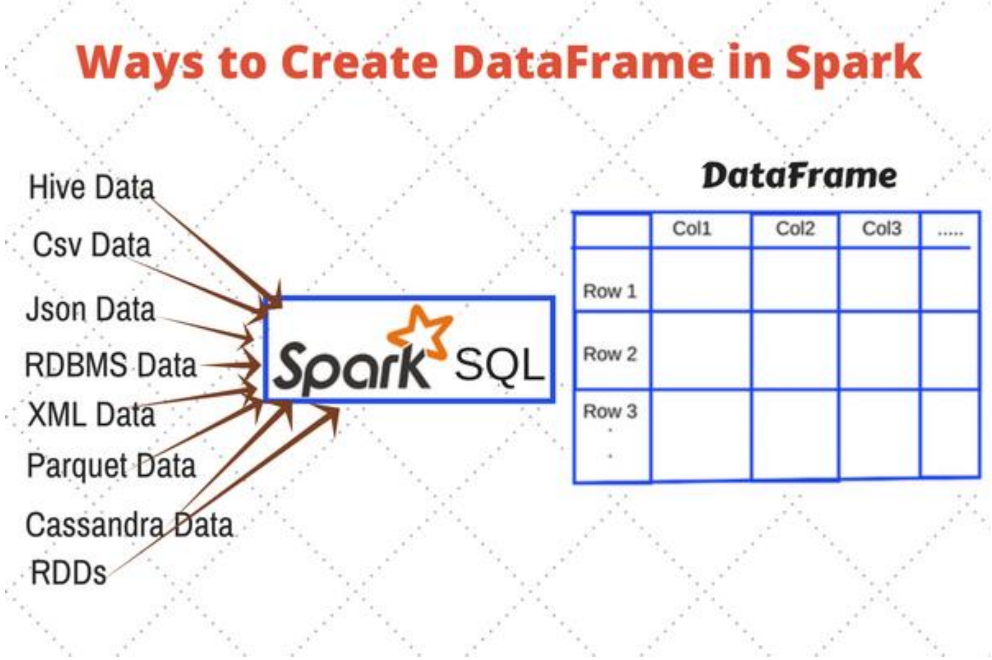


What is Serialization?

- Serialization is the process of translating data structures or objects state into binary or textual form to transport the data over network or to store on some persistent storage.
- Once the data is transported over network or retrieved from the persistent storage, it needs to be deserialized again.
- Serialization is termed as **marshalling** and deserialization is termed as **unmarshalling**.



DataFrame



- A Dataset is a distributed collection of data
- A DataFrame is a Dataset organized into named columns.
- It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. It is a distributed collection of rows with the same schema.
- Can be constructed from external data sources or RDDs into essentially an RDD of Row objects
- Supports relational operators (e.g. where, groupby) as well as Spark operations.
- Evaluated lazily ⇒ unmaterialized logical plan

Example of Dataframe

- **employee.json**

```
{
  {"id" : "1201", "name" : "satish", "age" : "25"}
  {"id" : "1202", "name" : "krishna", "age" : "28"}
  {"id" : "1203", "name" : "amith", "age" : "39"}
  {"id" : "1204", "name" : "javed", "age" : "23"}
  {"id" : "1205", "name" : "prudvi", "age" : "23"}
}
```

- Code to create the dataframe

```
scala> val dfs = sqlContext.read.json("employee.json")
dfs: org.apache.spark.sql.DataFrame = [age: string, id: string,
name: string]

scala> dfs.show()
+----+------+--------+
|age | id   |  name  |
+----+------+--------+
| 25 | 1201 | satish |
| 28 | 1202 | krishna|
| 39 | 1203 | amith  |
| 23 | 1204 | javed  |
| 23 | 1205 | prudvi |
+----+------+--------+
```

Data Model

- Spark SQL uses a nested data model based on Hive for tables and DataFrames

- Supports both primitive SQL types (boolean, integer, double, decimal, string, data, timestamp) and complex types (structs, arrays, maps, and unions); also user defined types.

- Complex data types can also be nested together to create more powerful types

- Accurately model data from a variety of sources and formats, including Hive, relational databases, JSON, and native objects in Java/Scala/Python

DataFrame Operations

- Relational operations (select, where, join, groupBy) via a domain-specific language (DSL) like SQL

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

- Operators take *expression* objects

- Operators build up an Abstract Syntax Tree (AST), which is then optimized by *Catalyst*.

- Alternatively, register as temp SQL table and perform traditional SQL query strings

```
users.where(users("age") < 21)
    .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

Advantages over Relational Query Languages

- DataFrames provide the same operations as relational query languages like SQL, we found that they can be significantly easier for users to work with thanks to their integration in a full programming language (control structures, e.g. if, for, etc.)

- Users can break up their code into Scala, Java or Python functions that pass DataFrames between them to build a logical plan

- Holistic optimization across functions composed in different languages.

- Logical plan analyzed eagerly $\Rightarrow$ identify code errors associated with data schema issues on the fly.

Querying Native Datasets

- To interoperate with procedural Spark code, Spark SQL allows users to construct DataFrames directly against RDDs of objects native to the programming language
- Infer column names and types directly from data objects (via reflection in Java and Scala and data sampling in Python, which is dynamically typed)

```
case class User(name: String , age: Int)

// Create an RDD of User objects

usersRDD = spark.parallelize(

List(User("Alice", 22), User("Bob", 19)))

// View the RDD as a DataFrame

usersDF = usersRDD.toDF
```

- Native objects accessed in-place to avoid expensive data format transformation.
- Benefits:
  - Run relational operations on existing Spark programs
  - Combine RDDs with external structured data

User-Defined Functions (UDFs)

- Easy extension of limited operations supported.
- Allows inline registration of UDFs.
- Can be defined on simple data types or entire tables.
- UDFs available to other interfaces (JDBC/ODBC for instance) after registration.
- Example:

```
val model: LogisticRegressionModel = ...
ctx.udf.register("predict",
    (x: Float, y: Float) => model.predict(Vector(x, y)))
ctx.sql("SELECT predict(age, weight) FROM users")
```

Spark SQL Guide

Datasets and DataFrames

- A Dataset is a distributed collection of data.
    - Dataset provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine.
    - The Dataset API is available in Scala and Java. Python does not have the support for the Dataset API.

- A DataFrame is a Dataset organized into named columns.
    - It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.
    - DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
    - The DataFrame API is available in Scala, Java, Python, and R. In Scala and Java, a DataFrame is represented by a Dataset of Rows.
    - In the Scala API, DataFrame is simply a type alias of Dataset[Row].

- Scala/Java Datasets of Rows are often referred as DataFrames.

Creating DataFrames

- Create a dataframe from a json file
  - val df = spark.read.json("/tmp/people.json")

- Displays the content of the DataFrame to stdout
  - df.show()

```
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

people.json

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

DataFrame Operations

- This import is needed to use the $-notation
  - import spark.implicits._
- Print the schema in a tree format
  - df.printSchema()

  // root

  // |-- age: long (nullable = true)

  // |-- name: string (nullable = true)

- A **schema** is the description of the structure of your data (which together create a dataset in Spark SQL).
  - It can be **implicit** (and inferred at runtime) or **explicit** (and known at compile time).
  - The implicit schema can be exact or approximative.
  - A schema is described using StructType which is a collection of StructField objects (that in turn are tuples of names, types, and nullability classifier).

Untyped Dataset Operations

- Select only the "name" column
  - df.select("name").show()

```
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

- Select everybody, but increment the age by 1
  - df.select($"name", $"age" + 1).show()

```
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+
```

- Select people older than 21
  - df.filter($"age" > 21).show()

```
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+
```

- Count people by age
  - df.groupBy("age").count().show()

```
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Running SQL Queries Programmatically

- Register the DataFrame as a SQL temporary view

  - df.createOrReplaceTempView("people")
  - val sqlDF = spark.sql("SELECT * FROM people")
  - sqlDF.show()

```
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Global Temporary View

- Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates.
- If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view.
- Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g. SELECT * FROM global_temp.view1.
- Register the DataFrame as a global temporary view
  - df.createGlobalTempView("people")
  - spark.sql("SELECT * FROM global_temp.people").show()

```
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Creating Datasets

- Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized encoder to serialize the objects for processing or transmitting over the network.
- While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.
- Define a case class
  - case class Person(name: String, age: Long)
- Encoders are created for case classes
  - val caseClassDS = Seq(Person("Andy", 32)).toDS()
  - caseClassDS.show()

```
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+
```

- The Case class in Scala is pretty much like a regular Scala class but with some additional functionality.
- The objects of this class can be instantiated even without using the “new” keyword.
- We can conveniently copy one object of the Case class to another entirely or even while changing some of the values of some of the attributes of this class.

Creating Datasets

- Encoders for most common types are automatically provided by importing spark.implicits._
    - val primitiveDS = Seq(1, 2, 3).toDS()
    - primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)

- DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name
    - val path = "/tmp/people.json"
    - val peopleDS = spark.read.json(path).as[Person]
    - peopleDS.show()

```
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Programmatically Specifying the Schema

- import org.apache.spark.sql.Row
- import org.apache.spark.sql.types._

// Create an RDD

- val peopleRDD = spark.sparkContext.textFile("/tmp/people.txt")

// The schema is encoded in a string

- val schemaString = "name age"

// Generate the schema based on the string of schema

- val fields = schemaString.split(" ")

  .map(fieldName => StructField(fieldName, StringType, nullable = true))

- val schema = StructType(fields)

// Convert records of the RDD (people) to Rows

- val rowRDD = peopleRDD

  .map(_.split(","))

  .map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD

- val peopleDF = spark.createDataFrame(rowRDD, schema)

// Creates a temporary view using the DataFrame

- peopleDF.createOrReplaceTempView("people")

// SQL can be run over a temporary view created using DataFrames

- val results = spark.sql("SELECT name FROM people")

// The results of SQL queries are DataFrames and support all the normal RDD operations

// The columns of a row in the result can be accessed by field index or by field name

- results.map(attributes => "Name: " + attributes(0)).show()

```
// +------------+
// |       value|
// +------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +------------+
```

Inferring the Schema from an RDD containing case classes

- The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame.
- The case class defines the schema of the table.
- This RDD can be implicitly converted to a DataFrame and then be registered as a table.
- Tables can be used in subsequent SQL statements.
- For implicit conversions from RDDs to DataFrames
  - import spark.implicits._
- Create a class to model Person
  - case class Person(name: String, age: Long)
- Create an RDD of Person objects from a text file, convert it to a Dataframe
  - val peopleDF = spark.sparkContext

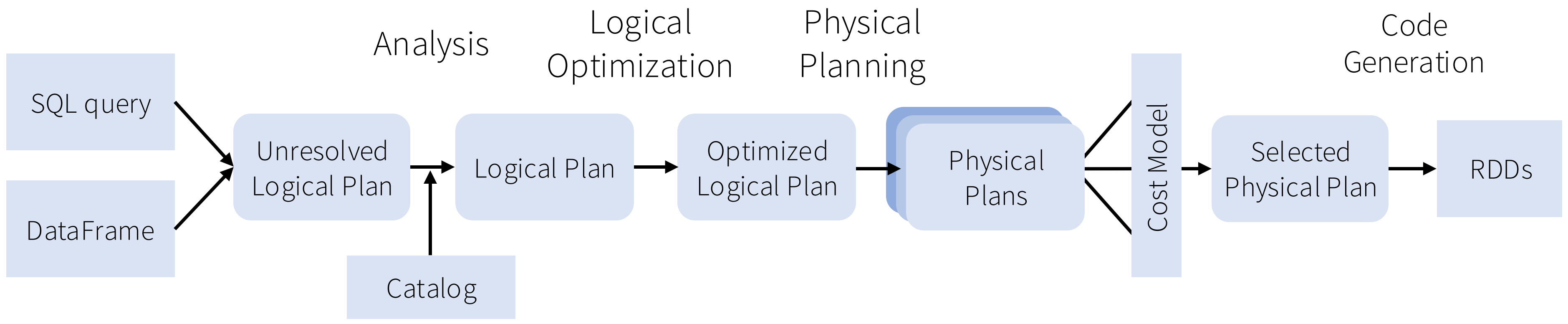
```
    .textFile("examples/src/main/resources/people.txt")
    .map(_.split(","))
    .map(attributes => Person(attributes(0), attributes(1).trim.toInt))
    .toDF()
```

- Register the DataFrame as a temporary view
  - peopleDF.createOrReplaceTempView("people")
- SQL statements can be run by using the sql methods provided by Spark
  - val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")
- The columns of a row in the result can be accessed by field index
  - teenagersDF.map(teenager => "Name: " + teenager(0)).show()

```
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

Catalyst

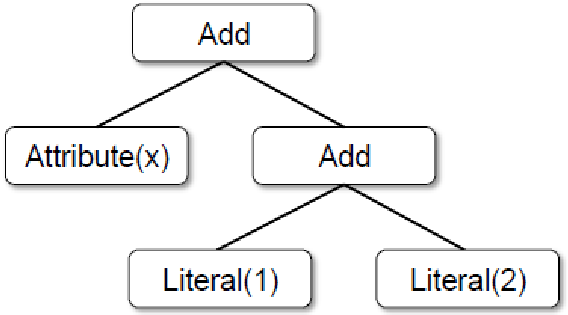
Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

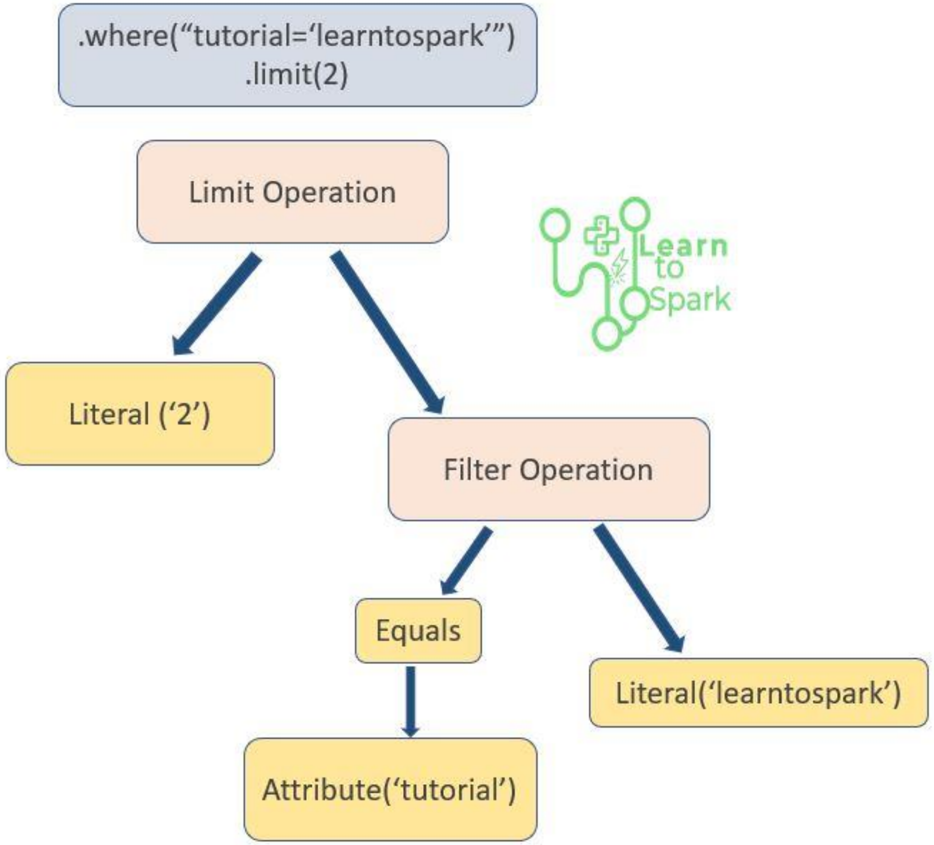
Trees

- The main data type in Catalyst is a tree composed of node objects.
- Each node has a node type and zero or more children. New node types are defined in Scala as subclasses of the TreeNode class.
- These objects are immutable and can be manipulated using functional transformations.
- Example:
  - Literal(value: Int): a constant value
  - Attribute(name: String): an attribute from an input row, e.g., “x”
  - Add(left: TreeNode, right: TreeNode): sum of two expressions.
  - Add(Attribute(x), Add(Literal(1), Literal(2)))



Example of Tree

- Expression: .where(“tutorial=‘learntospark’”).limit(2)

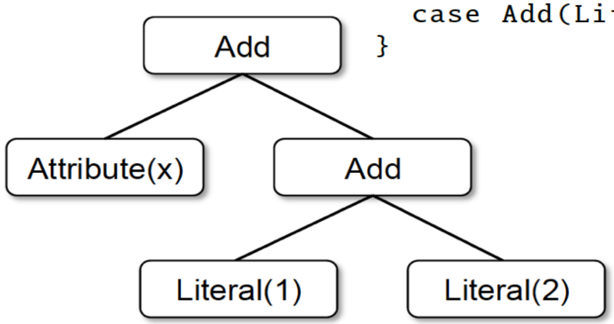


https://www.learntospark.com/2020/02/spark-sql-catalyst-optimizer.html

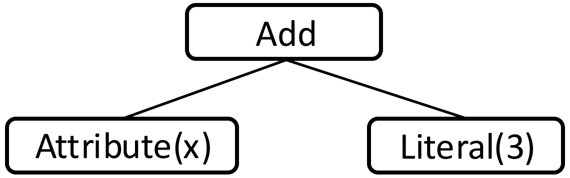
Catalyst Rules

- Pattern matching functions that transform subtrees into specific structures.
- Multiple patterns in the same transform call.
- May take multiple batches to reach a fixed point.
- Transform can contain arbitrary Scala code.

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

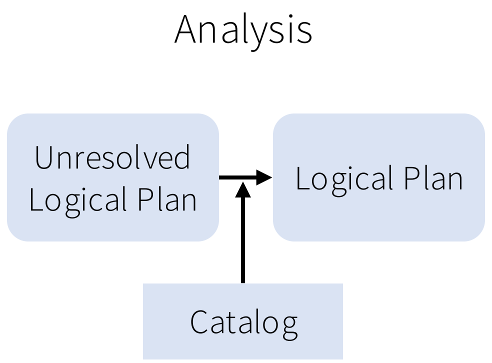


x + (1 + 2)



x + 3

Analysis



Example:

SELECT *col* FROM *sales*

The type of *col*, or even whether it is a valid column name, is not known until we look up the table *sales*.

- Spark SQL begins with a relation to be computed, either from a tree returned by a SQL parser, or from a DataFrame object constructed using the API
- An attribute is **unresolved** if its type is not known or it's not matched to an input table.
- The Catalog object that tracks the tables in all data sources
- To resolve attributes:
  - Look up relations by name from the catalog.
  - Map named attributes to the input provided given operator's children.
  - Identifier for references to the same value
  - Propagate and coerce types through expressions

Logical Optimization

Logical Plan → Optimized Logical Plan

Logical Optimization

- Applies standard rule-based optimization (constant folding, predicate-pushdown, projection pruning, null propagation, boolean expression simplification, etc.)

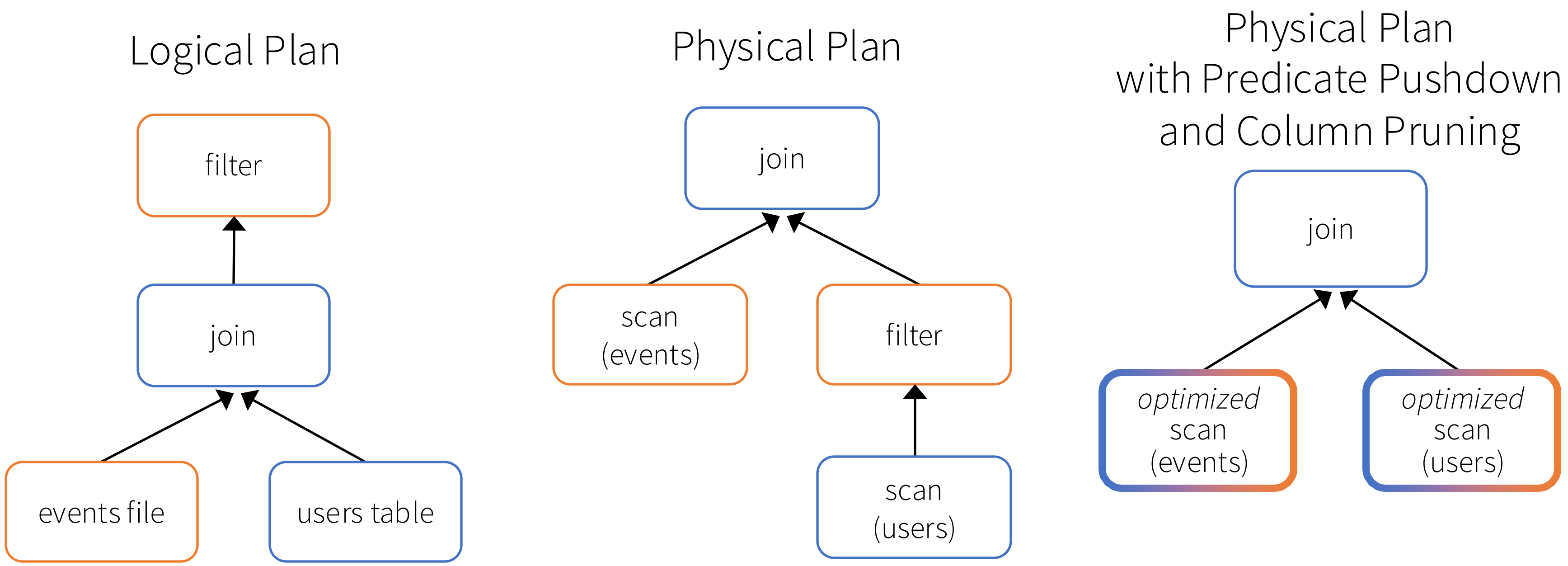
- Example: when the fixed-precision DECIMAL type were added to Spark SQL, it was wanted to optimize aggregations such as sums and averages on DECIMALs with small precisions; it took just a few lines of code to write a rule that finds such decimals in SUM and AVG expressions, and casts them to unscaled 64-bit LONGs, does the aggregation on that, then converts the result back.

```
object DecimalAggregates extends Rule[LogicalPlan] {
/** Maximum number of decimal digits in a Long */
val MAX_LONG_DIGITS = 18

def apply(plan: LogicalPlan): LogicalPlan = {
  plan transformAllExpressions {
    case Sum(e @ DecimalType.Expression(prec, scale))
         if prec + 10 <= MAX_LONG_DIGITS =>
       MakeDecimal(Sum(UnscaledValue(e)), prec + 10, scale)
  }
}
```

A simplified version of the code.

Physical Planning

```
def add_demographics(events):
    u = sqlCtx.table("users")                     # Load partitioned Hive table
    events.join(u, events.user_id == u.user_id)   # Join on user_id

events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne") # City is initially a field of "users"
                .select(events.timestamp).collect()
```



Physical Plan

- Physical plan is nothing but the conversion of the optimized logical plan into a physical plan that can be executed in the cluster.
- After the physical plans are generated, the cost is estimated recursively for the tree end-to-end.
- Finally, Spark uses the Cost Based Optimization to select the best suited physical plan to execute in cluster based on the provided data source.

Code Generation

- The final phase of query optimization involves generating Java bytecode to run on each machine.
- Catalyst relies on a special feature of the Scala language, quasiquotes, to make code generation simpler.
- Catalyst transforms a tree representing an expression in SQL to an Abstract Syntax Tree (AST) for Scala code to evaluate that expression, and then compile and run the generated code.
- The strings beginning with q are **quasiquotes**, meaning that although they look like strings, they are parsed by the Scala compiler at compile time.
- With code generation, we can write a function to translate a specific expression tree to a Scala AST as follows:

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

SparkSQL Conclusion

SparkSQL Conclusion

- Let developers create and run Spark programs faster:
  - Write less code
  - Read less data
  - Let the optimizer do the hard work

- DataFrames and SQL provide a common way to access a variety of data sources

- Lots of extensions by using User Defined Functions