

# SPARK SQL

# SPARK DATA FRAMES

Mahidhar

Reference : [http://people.csail.mit.edu/matei/papers/2015/sigmod\\_spark\\_sql.pdf](http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf)



# SPARK SQL

- Spark SQL provides support for loading and manipulating structured data in Spark, either from external structured data sources or by adding a schema to an existing RDD.
- Spark introduced the major component, Spark SQL, for loading and manipulating structured data in Spark with Spark 1.0 version.
- Spark SQL's API interoperates with the RDD data model, allowing users to interleave Spark code with SQL statements.
- Under the hood, Spark SQL uses the Catalyst optimizer to choose an efficient execution plan, and can automatically push predicates into storage formats like Parquet.



## contd..

- Spark SQL is a Spark module for structured data processing.
- Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed.
- There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result the same execution engine is used, independent of which API/language you are using to express the computation.
- This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.



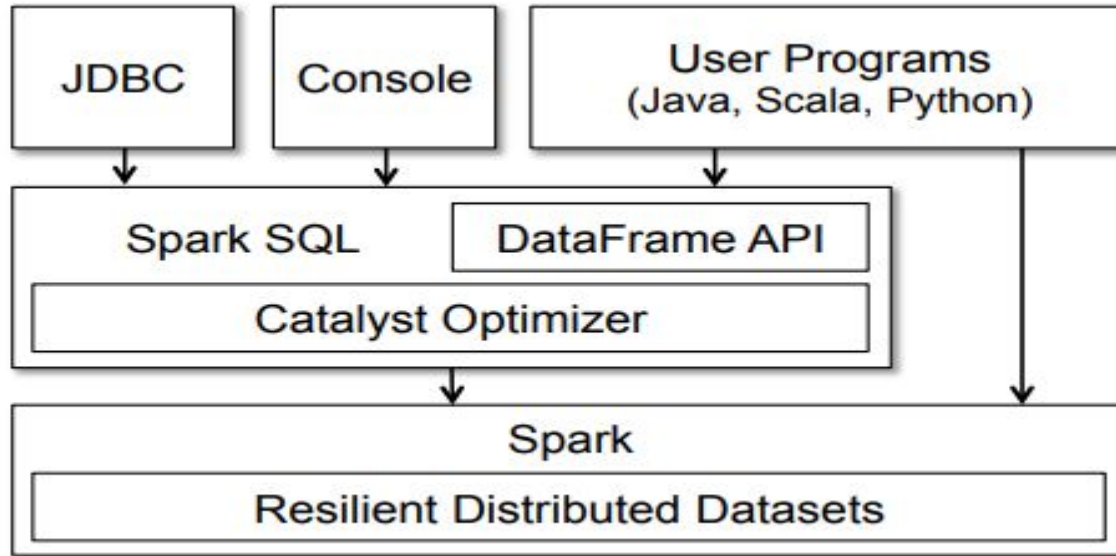
## contd..

- One use of Spark SQL is to execute SQL queries.
- When running SQL from within another programming language the results will be returned as a DataFrame.
- You can also interact with the SQL interface using the command-line or over JDBC/ODBC.
- `spark-sql > CREATE TABLE t1 (i INT);`
- We can run the same command from the jupyter notebook using the

`spark.sql("""CREATE TABLE t1 (i INT)""")` after creating the spark session by name spark , which will create a data frame.

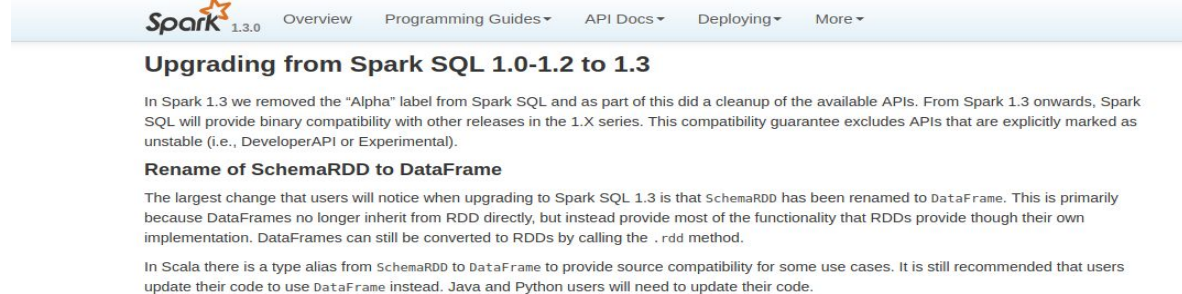


# Interfaces to Spark SQL, and interaction with Spark



# Data Frame

- A Data Frame is an immutable distributed collection of data that is organized into named columns analogous to a table in a relational database
- It was introduced as experimental feature within Apache Spark 1.0 as SchemaRDD and later with Spark 1.3 release they were renamed to Data Frames



The screenshot shows the Apache Spark 1.3.0 documentation page. The navigation bar includes links for Overview, Programming Guides, API Docs, Deploying, and More. The main heading is "Upgrading from Spark SQL 1.0-1.2 to 1.3". The text explains that in Spark 1.3, the "Alpha" label was removed from Spark SQL, and a cleanup of APIs was performed. It states that Spark SQL now provides binary compatibility with other releases in the 1.X series, excluding APIs marked as unstable (DeveloperAPI or Experimental). A sub-section titled "Rename of SchemaRDD to DataFrame" details the changes: SchemaRDD has been renamed to DataFrame, and DataFrames no longer inherit from RDD directly. It also mentions that DataFrames can still be converted to RDDs using the .rdd method. Finally, it notes that in Scala, there is a type alias from SchemaRDD to DataFrame for source compatibility, and that Java and Python users will need to update their code.

**Upgrading from Spark SQL 1.0-1.2 to 1.3**

In Spark 1.3 we removed the "Alpha" label from Spark SQL and as part of this did a cleanup of the available APIs. From Spark 1.3 onwards, Spark SQL will provide binary compatibility with other releases in the 1.X series. This compatibility guarantee excludes APIs that are explicitly marked as unstable (i.e., DeveloperAPI or Experimental).

**Rename of SchemaRDD to DataFrame**

The largest change that users will notice when upgrading to Spark SQL 1.3 is that SchemaRDD has been renamed to DataFrame. This is primarily because DataFrames no longer inherit from RDD directly, but instead provide most of the functionality that RDDs provide through their own implementation. DataFrames can still be converted to RDDs by calling the `.rdd` method.

In Scala there is a type alias from SchemaRDD to DataFrame to provide source compatibility for some use cases. It is still recommended that users update their code to use DataFrame instead. Java and Python users will need to update their code.

## contd..

- 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.
- Once constructed, they can be manipulated with various relational operators, such as `where` and `groupBy`, which take expressions in a domain-specific language (DSL) similar to data frames in R and Python.
- The DataFrame object represents a logical plan to compute a dataset, but no execution occurs until the user calls a special “**output operation**” such as `save` is called upon it.
- Data Frame transformations are also **lazy**.



# Simple SPARK-SQL Query

- ❖ `spark.sql("""create database insofe_mahidhar """)`
- ❖ `spark.sql("""use insofe_mahidhar""")`
- ❖ `users = spark.sql (""" create table users .... """)`
- ❖ `young = users.filter(users. age < 25)`
- ❖ `print(young.count ())`



- When the user calls count, which is an output operation, Spark SQL builds a physical plan to compute the final result. This might include optimizations such as only scanning the "age" column of the data if its storage format is columnar, or even using an index in the data source to count the matching rows.





## contd..

- It supports all major SQL data types, including boolean, integer, double, decimal, string, date, and timestamp, as well as complex (i.e., non-atomic) data types: structs, arrays, maps and unions.
- Users can perform relational operations on DataFrames using a domain-specific language (DSL) similar to R data frames and Python Pandas .
- DataFrames support all common relational operators, including projection (select), filter (where), join, and aggregations (groupBy).
- These operators all take expression objects in a limited DSL that lets Spark capture the structure of the expression.
- In Pyspark , Spark SQL samples the dataset to perform schema inference due to the dynamic type system.

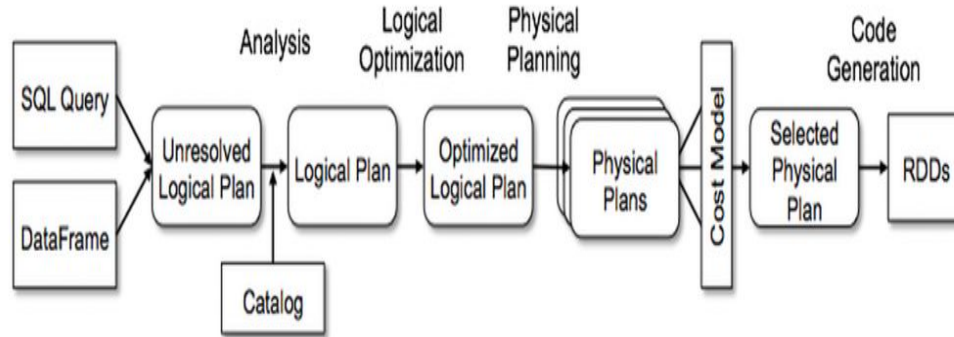


## contd..

- Code which is written in Structured APIs is always converted into Low level APIs internally. In programming, one can navigate between Higher i.e. Structured APIs to Lower level APIs and vice versa. (DataFrame -----> rdd using .rdd on Data Frames) and (rdd -----> toDF() on RDD)
- Using the Structured APIs one can leverage application in following ways –
  - Code in Structured APIs like Datasets, DataFrames, SQL
  - If valid code, Spark converts it into a Logical Plan
  - Spark internal transformation converts Logical Plan to Physical Plan
  - Spark then executes this Physical Plan on cluster



# Phases of query planning in Spark SQL



- analyzing a logical plan to resolve references
- logical plan optimization
- physical planning, and
- code generation to compile parts of the query to Java bytecode

# Unresolved Logical Plan

It starts by building an “unresolved logical plan” tree with unbound attributes and data types, then applies rules that do the following:

- Looking up relations by name from the catalog.
- Mapping named attributes, which are referred as col .
- Determining which attributes refer to the same value to give them a unique ID (which later allows optimization of expressions such as col = col).
- Propagating and coercing types through expressions:

for example: we cannot know the type of  $1 + \text{col}$  until we have resolved col and possibly cast its subexpressions to compatible types.

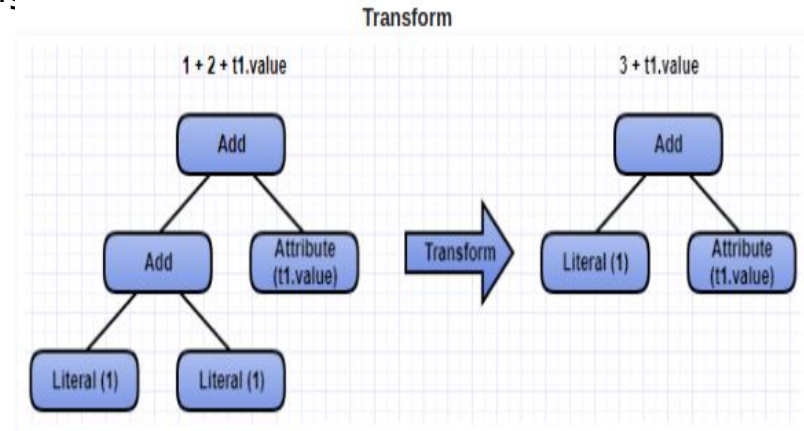


# Logical Plan

- Logical Plan describes computation on datasets without defining how to conduct the computation
- The logical plan optimization applies rule-based optimizations to logical plan.
- These includes constant folding etc.

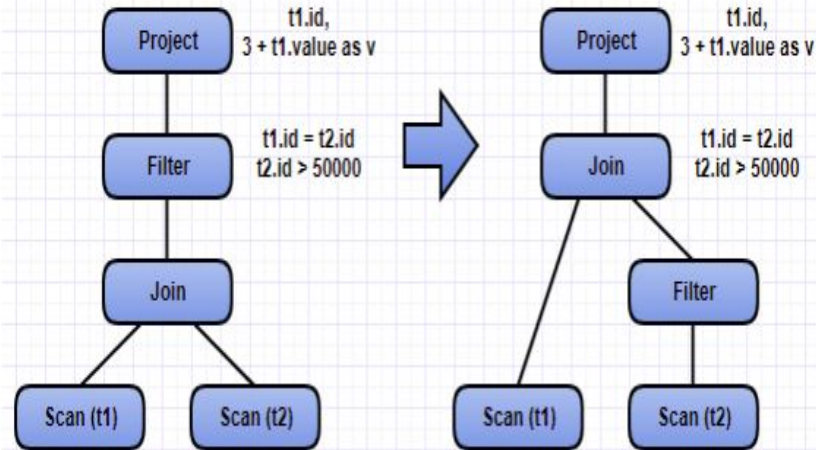
Ex : 

```
SELECT sum(v)
FROM(
  SELECT
    t1.id ,
    1+2+t1.values AS v
  FROM t1 JOIN t2
  WHERE
    T1.id = t2.id AND
    T2.id > 50000 ) tmp
```

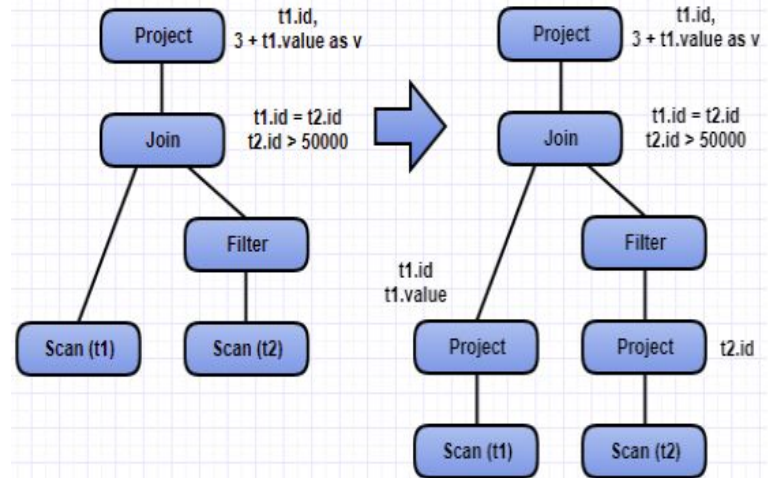


# Contd ..

Predicate Pushdown



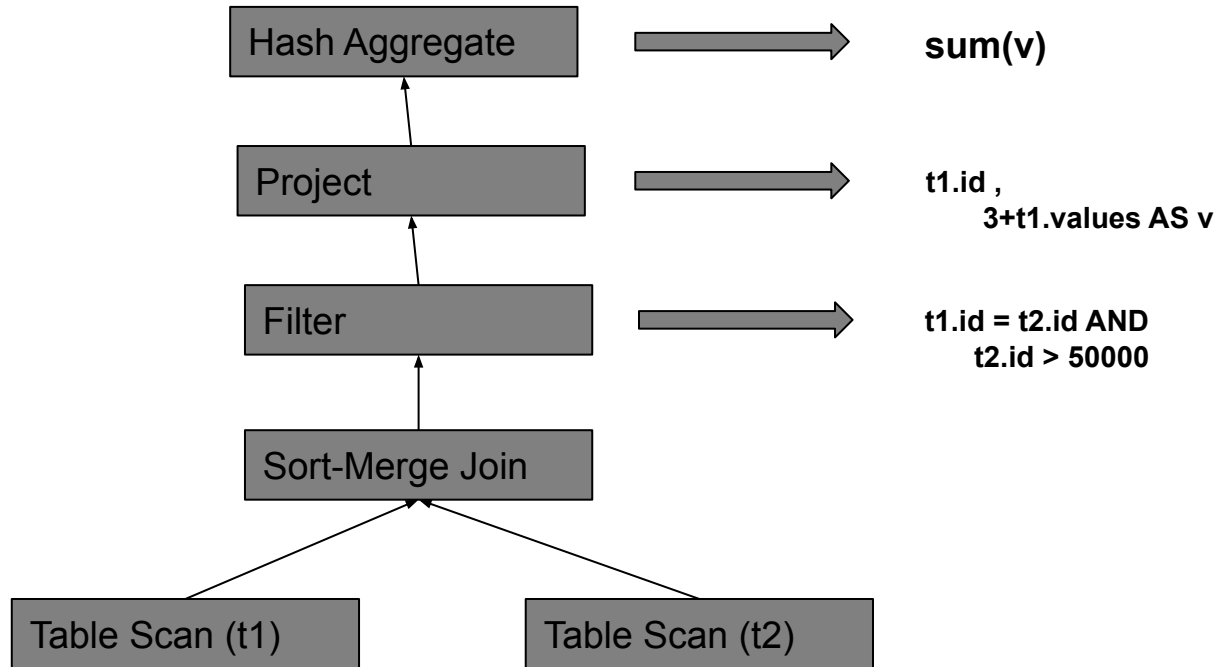
Column Pruning



# Physical Plan

- A physical plan describes the actual computation with specific definition of how to compute it.
- In the physical planning phase, Spark SQL takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine.
- It then selects a plan using a cost model.
- At the moment, cost-based optimization is only used to select join algorithms:
  - for relations that are known to be small, Spark SQL uses a broadcast join, using a peer-to-peer broadcast facility available in Spark.







# Code Generation

- The final phase of query optimization involves generating Java bytecode to run on each machine.
- Because Spark SQL often operates on in-memory datasets, where processing is CPU-bound, we wanted to support code generation to speed up execution.
- Nonetheless, code generation engines are often complicated to build, amounting essentially to a compiler. Catalyst relies on a special feature of the Scala language, quasiquotes, to make code generation simpler.



# Tungsten Engine

- Project Tungsten will be the largest change to Spark's execution engine since the project's inception. It focuses on substantially improving the efficiency of *memory and CPU* for [Spark applications](#), to push performance closer to the limits of modern hardware.
- It emits optimized bytecode at runtime that collapses the entire query into a single function, eliminating virtual function calls and leveraging CPU registers for intermediate data.
- As a result of this streamlined strategy, called “whole-stage code generation,” we significantly improve CPU efficiency and gain performance.

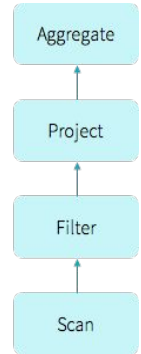


# Contd..

- Previously Volcano Model has been used with the spark 1.6 and later they introduced the Second generation Tungsten engine with "Whole Stage Code Generation"
- The Query on the right hand side uses the next() of the Volcano Model and call it multiple times.
- Spark came up removing this Virtual Function calls by generating code that is suitable for the exact requirement. Which is like a normal loop for the query

```
var count = 0
for (ss_item_sk in store_sales) {
    if (ss_item_sk == 1000) {
        count += 1
    }
}
```

```
select count(*) from store_sales
where ss_item_sk = 1000
```



- **Observation is no virtual function calls**

# Contd..

## Intermediate data in memory vs CPU registers:

- In the Volcano model, each time an operator passes a tuple to another operator, it requires putting the tuple in memory (function call stack).
- In the hand-written version, by contrast, the compiler (JVM JIT in this case) actually places the intermediate data in CPU registers. Again, the number of cycles it takes the CPU to access data in memory is orders of magnitude larger than in registers.

## Loop unrolling and SIMD:

- Modern compilers and CPUs are incredibly efficient when compiling and executing simple for loops. Compilers can often unroll simple loops automatically, and even generate SIMD instructions to process multiple tuples per CPU instruction.
- CPUs include features such as pipelining, prefetching, and instruction reordering that make executing simple loops efficient. These compilers and CPUs, however, are not great with optimizing complex function call graphs, which the Volcano model relies on.



## When to use RDDs?

Consider these scenarios or common use cases for using RDDs when:

- you want low-level transformation and actions and control on your dataset;
- your data is unstructured, such as media streams or streams of text;
- you want to manipulate your data with functional programming constructs than domain specific expressions;
- you don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and
- you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.



## When should I use DataFrames or Datasets?

- If you want rich semantics, high-level abstractions, and domain specific APIs, use DataFrame or Dataset.
- If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data, use DataFrame or Dataset.
- If you want higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation, use Dataset.
- If you want unification and simplification of APIs across Spark Libraries, use DataFrame or Dataset.
- If you are a R user, use DataFrames.
- If you are a Python user, use DataFrames and resort back to RDDs if you need more control.

