Spark Dataframe API and Spark SQL

pm jat @ daiict

Further Higher Abstractions

- **RDD are still "low level" to perform analytical tasks**!
- Spark provides higher level abstractions
  - "Dataframe" API (more "**structured RDDs**")
  - Spark-SQL (SQL interface)
- These abstractions makes "cluster programming" amazingly simple, and
- Primarily the reason Spark is becoming popular for big data processing.

SQL Interface over huge raw files!

- We require SQL interface over raw data file, so that, we are able to run queries without loading data into some database systems
  – For many ETL operations, load time into database systems is forbiddingly high
  – Schema check happens only when we run the query (Read time schema validation)
- This makes running ad-hoc queries on data files quick
- This has been motivation for Pig, Hive, and now Spark-SQL

Before Spark SQL

- Hive from Facebook
- Pig from Yahoo
  - Not really SQL but a scripting language like perl
- Cloudera Impala
- Google Dremel

Hive and Pig

- **Hive**: SQL like interface for HDFS files. Initially developed at Facebook (now Apache project).

```
SELECT count(*) FROM users
```

In reality, 90+% of MR jobs are generated by Hive SQL

- **Pig**: scripting language for various data transformations. Initially developed at Yahoo. Now again apache project

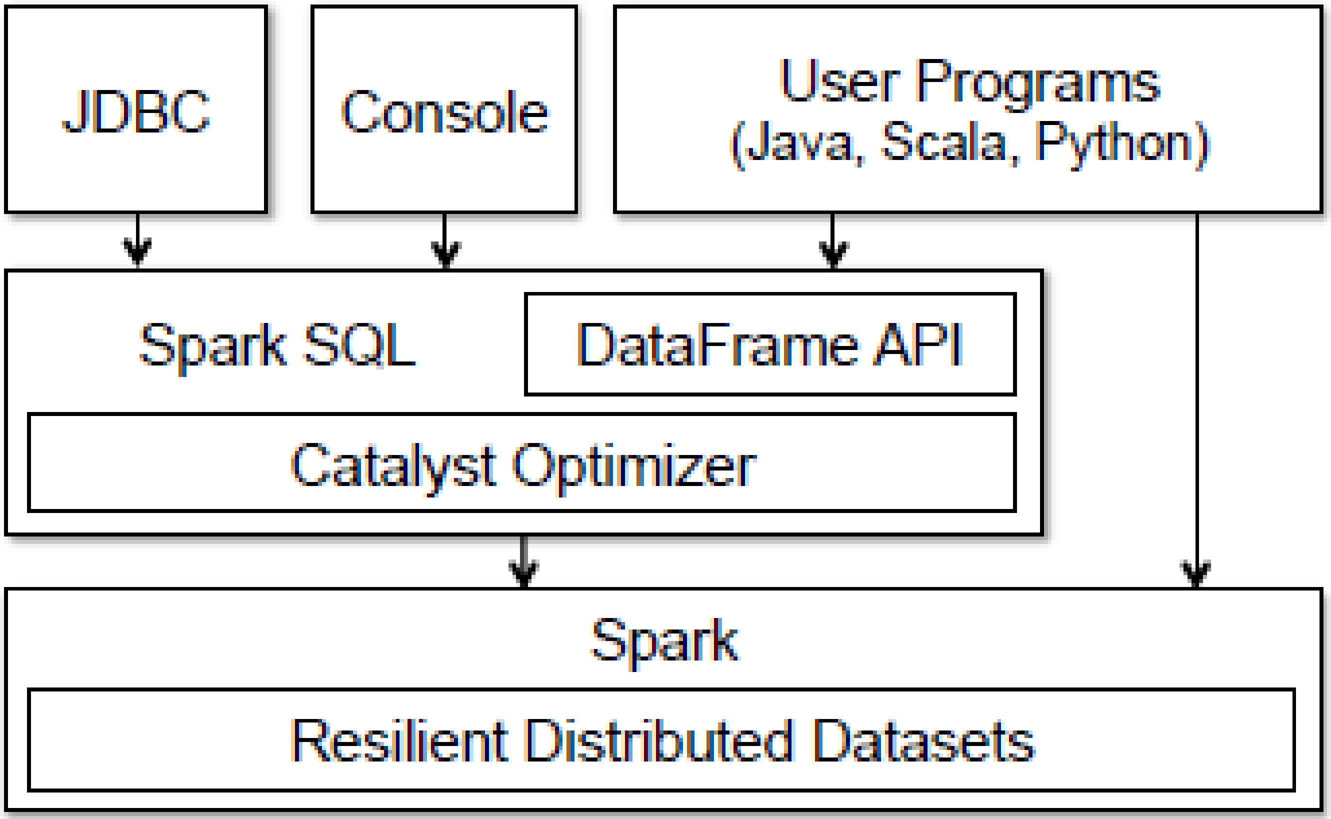
```
A = load 'foo';
B = group A all;
C = foreach B generate COUNT(A);
```

https://rxin.github.io/talks/2017-12-05_cs145-stanford.pdf

Spark-SQL

- Spark SQL was introduced in 2015 through paper "Spark SQL: Relational data processing in spark." in ACM SIGMOD
- Initial effort to have SQL interface over Spark was **Shark**, and was basically an adaption of "Hive over Spark".
- Shark, however
  - could not integrate well will with intermediate RDD datasets, and
  - Secondly Hive optimizer could not properly adapted to Spark as it was primarily designed for Map Reduce.
- And, we have Spark-SQL[1, 2015]

Spark-SQL – Stack[1]



[1] Armbrust, Michael, et al. "Spark sql: Relational data processing in spark." *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015

Spark-SQL - key characteristics

- Spark-SQL has following key characteristics-
  - Performance (since underlying engine is Spark)
  - Ability to access “procedures” from SQL statements
    - Opens up gate (for SQL) to interact with ML, Graph Processing algorithms as “User Defined Functions”
  - Query Optimizer
  - Select regmodel.predictsalary(resume) from employee;

Spark SQL - Programming Interface

- Spark SQL runs as a library on top of Spark (refer stack diagram)
- It has become buzzword for "Declarative Big Data Processing"
  – Write less code
  – Optimize the execution
- Spark exposes SQL interfaces, which can be accessed through
  – Command Line Interface (Console)
  – DataFrame API integrated into Spark programs
  – JDBC/ODBC access (through Spark Thrift Server)

Spark "Data Frame API" Source: [1]

- The main abstraction in Spark SQL's API is a DataFrame, a "distributed collection of rows with a **homogeneous schema**".
- A DataFrame is equivalent to a table in a relational database, and can also be manipulated in similar ways.
- Unlike RDDs, DataFrames keep track of their schema and support various relational operations on them.
- DataFrames can be constructed from data files, tables in a system catalog (based on external data sources) or from existing RDD objects!

Transformations and Actions on DataFrame Source: [2]

Transformation examples	Action examples
• filter	• count
• select	• collect
• drop	• show
• intersect	• head
• join	• take

- *Note:* Dataframe evaluations are lazy. Transformations are just getting expressed (not executed immediately). It the request of actions that lead to real execution (and this is supported by optimization)

DataFrame transformation are Lazy

Source: [1]

- Unlike traditional (like in Panda and R) data frame APIs, Spark Data Frames are lazy,
  - in spark, each DataFrame object represents a logical plan to compute a dataset frame
- It is like, when specified, it adds to execution path, and executed only when some action is to be performed.
- This enables optimization across all operations that were used to build the Data Frame.

Data Frames and RDDs

- DataFrames are built on top of the RDD and are "structured", that is have schema attached with them.
- We can call them Schema aware RDDs - element type is "Row with Schema"
- Also immutable
- DataFrames are more structured, provides abstraction similar to a database table. This makes it very simple to program.
- Dataframes are promised to be more efficient, because operations on them can be optimized through "Catalyst", the optimizer for Spark-SQL!

Programming Dataframes

Spark Session

- Prior to Spark 2.0, Spark had three Contexts
  - Spark Context: for RDD operations
  - SQL Context: for Spark SQL operations
  - Hive Context: for Hive queries
- However Spark 2.0 onwards, there is a unified context called Spark Session, all operations can be performed through spark session, therefore
- Spark Session is more recent!

Initialize Spark Session

- Before doing anything on Spark Dataframe API or Spark SQL we require creating Spark Session

- Typically done as following

```
SparkSession spark = SparkSession.builder()
          .appName("Spark Demo")
          .master("local")
          .getOrCreate();
```

Reading from a CSV file as Dataframe

- Typically done as following:

```
Dataset<Row> emp = spark.read()
        .option("header", "true")
        .option("sep", ",")
        .csv("data/employee_m.csv");
```

- Peek into read data rows Can from the CSV file.

```
//show all rows
emp.show(); //select * from emp;
//show first 3 rows
emp.show(3); //select * from emp limit 3;
```

Reading from a CSV file as Dataframe

- By default
  - If No Header Row in data, then columns are named as _c0, _c1, _c2, and so on
  - To indicate that file has header row, we say: `.option("header", "true")`
  - Schema remains undefined. By default data type for each column remains string.
- We can ask spark to automatically infer the schema, or we can manually supply the schema before reading a CSV file.

Inferring Schema

- Set : `.option("inferSchema", "true");` to auto infer the schema (in read options)
- We can dump the inferred schema as following:
  `emp.printSchema();`

```
root
 |-- eno: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- dob: timestamp (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)
 |-- sup_eno: integer (nullable = true)
 |-- dno: integer (nullable = true)
```

- For inferring the schema, processor require scanning full file. May be expensive when file is huge, and may not always be successful.

Manually Specifying the Schema

(1) Define Schema:

```
StructType schema = DataTypes.createStructType(new StructField[] {
        //eno,name,dob,gender,salary,sup_eno,dno
    DataTypes.createStructField("eno", DataTypes.StringType, false),
    DataTypes.createStructField("name", DataTypes.StringType, true),
    DataTypes.createStructField("dobb", DataTypes.DateType, true),
    DataTypes.createStructField("gender", DataTypes.StringType, true),
    DataTypes.createStructField("salary", DataTypes.IntegerType, true),
    DataTypes.createStructField("sup_eno", DataTypes.StringType, true),
    DataTypes.createStructField("dno", DataTypes.IntegerType, true)
});
//System.out.println(schema.prettyJson());
```

(2) Specify Schema:

```
Dataset<Row> emp = spark.read()
        .option("header", "true")
        .option("sep", ",")
        .schema(schema)
        .csv("data/employee_m.csv");
```

Reading from a JSON file as Dataframe

- Data frame object is constructed and used as following:

```
Dataset<Row> emp1 = spark.read()
                    .json("data/employees.json");
emp1.printSchema();
emp1.show();
```

```
root
 |-- name: string (nullable = true)
 |-- salary: long (nullable = true)
```

```
+-------+------+
|   name|salary|
+-------+------+
|Michael|  3000|
|   Andy|  4500|
| Justin|  3500|
|  Berta|  4000|
+-------+------+
```

Examples – Filter and Sort

Filter (/where)

```
emp.filter( "salary > 30000" ).show();

emp.where( "salary > 40000" ).show();

emp.filter( emp.col("salary").geq(40000)
.and(emp.col("dno").equalTo(5))).show(); //More TYPE SAFE
```

Sort/Order By

```
emp.sort(emp.col("dno"),emp.col("salary").desc())
.show();

emp.orderBy(emp.col("dno"),emp.col("salary"))
.show();
```

Examples - Project

Project

```
emp.select("eno", "name", "salary")
      .where("dno == 4")
      .show(); //SELECT eno, name, salary from emp

                  //where dno=4

emp.select(
      emp.col("eno"),
      emp.col("name"),
      emp.col("salary").multiply(1.1)
).where("dno == 4")
.show();
```

Examples - Aggregation

Aggregation and Group By

```
//on whole Table
emp.agg(avg(emp.col("salary")),max(emp.col("salary")))
.show();

//Group By and Aggregation
emp.groupBy(emp.col("dno"))
.agg(avg(emp.col("salary")), max(emp.col("salary")))
.show();
```

Examples - Join

```
//Join and Project
Dataset<Row> empdep =
emp.join(dep, emp.col("dno").equalTo(dep.col("dno")));

empdep.select(emp.col("name"), dep.col("name"),
                                        emp.col("salary"))
.show();


//Join and Aggregate
emp.join(dep, emp.col("dno").equalTo(dep.col("dno")))
    .groupBy(dep.col("name"))
    .agg(sum(emp.col("salary")))
.show();
```

[Python] Constructing Data Frame

Source: [2]

```python
# Construct a DataFrame from a "users" table in Hive.
df = sqlContext.table("users")

# Construct a DataFrame from a log file in S3.
df = sqlContext.load("s3n://someBucket/path/to/data.json", "json")
```

[Python] Operation on Data Frames

Source: [2]

```
# Create a new DataFrame that contains only "young" users
young = users.filter(users["age"] < 21)

# Alternatively, using a Pandas-like syntax
young = users[users.age < 21]

# Increment everybody's age by 1
young.select(young["name"], young["age"] + 1)

# Count the number of young users by gender
young.groupBy("gender").count()

# Join young users with another DataFrame, logs
young.join(log, logs["userId"] == users["userId"], "left_outer")
```

[Python] Example: DataFrame API

- Let us say a tab separated data file called "**SalesProduct.txt**", where attributes Name, and Weight at 2nd and 8th position respectively.
- Following Spark program (using dataframe API) lists Name and Weight of top 15 products in the descending order weight!

```
sqlContext = SQLContext(sc)
```

```
rdd1 = (content.filter(lambda line: line.split("\t")[7] != "NULL")
 .map(lambda line: (line.split("\t")[1], float(line.split("\t")[7])))
)
```

```
df = sqlContext.createDataFrame(rdd1, schema = ["Name", "Weight"])
```

```
df.orderBy("weight", ascending = False).show(15, truncate = False)
```

Spark SQL

Spark SQL

- Spark SQL allows you to manipulate distributed data with SQL queries.
- That is we can execute an SQL statement on a Dataframe object. Though this requires us registering a dataframe as View using `createOrReplaceTempView()`
- Dataframe object provides SQL method for executing SQL statements. This method always returns a dataframe object.
- We can mix DataFrame methods and SQL queries in the same code.

Spark SQL

- Concepts are very similar to relational “Table” and “SQL”

Tables

- In programmer’s perspective table is identical to relational table.
- Here tables can be temporary or be stored.
- We typically have one data file for one table.
- Table data may be stored in “storage formats” that helps in efficient execution of queries.

Spark SQL

- Spark SQL provides same set of commands **DDL + DML**
- However commands do have some additional parameters that are specific to distributed data and clustered computing
  – For instance partitioning, shuffling, and sorting strategy, etc.
- Spark SQL run-time system provides “Spark SQL Engine” that “efficiently” executes Spark-SQL operations.
- SQL statements can be executed from CLI (command level interface) or host programs.
- CLI allows executing SQL commands interactively.

Spark SQL statements

- CREATE TABLE - has options like PARTITION BY, CLUSTER BY
  - Also ALTER TABLE, DROP TABLE, etc
- INSERT, UDPATE, LOAD data,
- CREATE, DROP FUNCTION
- Various commands to add, merge, delete in “data lakes”. Data lake is basically large scale data repository. One can be spark based data lake with SQL interface.
- Here is SQL reference manual from Databrick https://docs.databricks.com/spark/latest/spark-sql/index.html

CREATE table

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  USING datasource
  [OPTIONS (key1=val1, key2=val2, ...)]
  [PARTITIONED BY (col_name1, col_name2, ...)]
  [CLUSTERED BY (col_name3, col_name4, ...) INTO num_buckets BUCKETS]
  [LOCATION path]
  [COMMENT table_comment]
  [TBLPROPERTIES (key1=val1, key2=val2, ...)]
  [AS select_statement]
```

CREATE table options

- USING
  - Data file storage format
  - Can be one of - TEXT, CSV, JSON, JDBC, PARQUET, ORC, HIVE, DELTA, and LIBSVM
- Option PARTITIONED BY
  - Partition the created table by the specified columns.
- Option CLUSTERED BY
  - Each partition in the created table will be split into a fixed number of buckets by the specified columns. This is typically used within partition for minimizing read time.

SELECT statement

```
SELECT [hints, ...] [ALL|DISTINCT] named_expression[, named_expression, ...]
  FROM relation[, relation, ...]
  [lateral_view[, lateral_view, ...]]
  [WHERE boolean_expression]
  [aggregation [HAVING boolean_expression]]
  [ORDER BY sort_expressions]
  [CLUSTER BY expressions]
  [DISTRIBUTE BY expressions]
  [SORT BY sort_expressions]
  [WINDOW named_window[, WINDOW named_window, ...]]
  [LIMIT num_rows]
```

SELECT statement

- Option DISTRIBUTE BY
  - Repartition rows in the relation based on a set of expressions. Rows with the same expression values will be hashed to the same worker
- Option CLUSTER BY (WITHIN Partition)
  - Repartition rows in the relation based on a set of expressions and sort the rows in ascending order based on the expressions.
- Also have commands like SAMPLE, ROLLUP, CUBE, etc

Spark SQL Examples

- Suppose, we already have a dataset object emp constructed and read some data.
- Below is how we can execute a SQL statement on this. Details of result dataset are printed, and output is shown here.

```
emp.createOrReplaceTempView("employee");
String sql = "SELECT eno, name, salary FROM employee WHERE dno=4";
Dataset<Row> emp_dno4 = spark.sql( sql );
emp_dno4.show();
emp_dno4.printSchema();
```

```
+---+--------+------+
|eno|    name|salary|
+---+--------+------+
|106|Jennifer| 43000|
|107|   Ahmad| 25000|
|108|  Alicia| 25000|
+---+--------+------+
```

```
root
 |-- eno: string (nullable = true)
 |-- name: string (nullable = true)
 |-- salary: integer (nullable = true)
```

Example: Registering Tables

```
SparkSession spark = SparkSession.builder()
        .appName("Spark Demo").master("local").getOrCreate();

Dataset<Row> emp = spark.read()
        .option("header", "true")
        .option("sep", ",")
        .option("inferSchema", "true")
        .csv("data/employee_m.csv");

Dataset<Row> dep = spark.read()
        .option("header", "true")
        .option("sep", ",")
        .option("inferSchema", "true")
        .csv("data/department_m.csv");

emp.createOrReplaceTempView("employee");
dep.createOrReplaceTempView("department");
```

Spark SQL Examples

- Various SQL operations on Employee and Department tables. Code should be self explanatory!

```
String sql = "SELECT d.name, e.name FROM employee e JOIN "
        + "department d ON (e.dno=d.dno)";
spark.sql( sql ).show();

sql = "SELECT e.name, s.name FROM employee e LEFT JOIN "
        + "employee s ON (e.sup_eno=s.eno)";
spark.sql( sql ).show();

sql = "SELECT dno, sum(salary) as TotalSal FROM employee "
        + "group by dno order by sum(salary) desc ";
spark.sql( sql ).show();
```

(Python) Example: Spark SQL

```
sqlContext = SQLContext(sc)
```

```
rdd1 = (content.filter(lambda line: line.split("\t")[7] != "NULL")
 .map(lambda line: (line.split("\t")[1], float(line.split("\t")[7])))
)
```

```
df = sqlContext.createDataFrame(rdd1, schema = ["Name", "Weight"])
```

```
df.createOrReplaceTempView("df_table")
```

```
sqlContext.sql(" SELECT * FROM df_table  ORDER BY Weight DESC limit 15").show()
```

Spark SQL – Procedural integration[1]

- Spark SQL can seamlessly integrate with procedures through its concept of User Defined Functions!

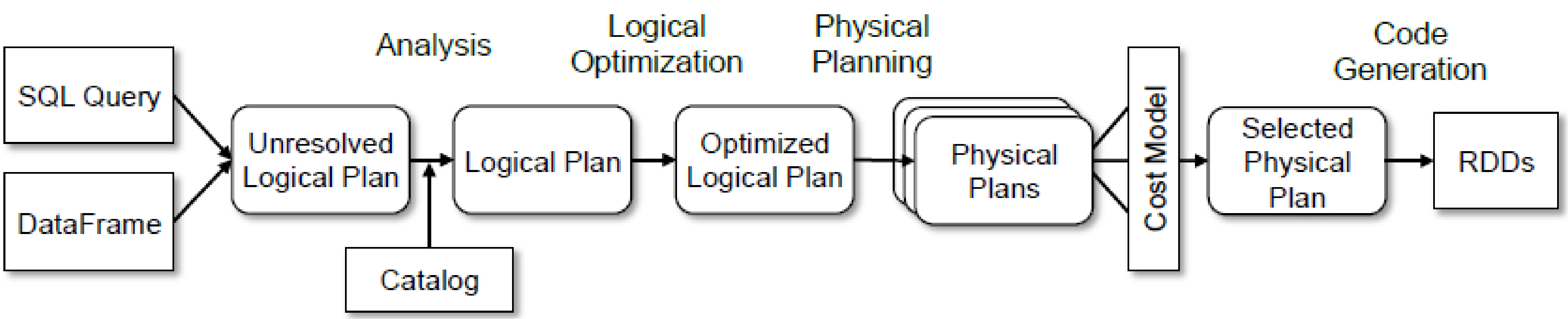
- Below is Scala example

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

[2] Armbrust, Michael, et al. "Spark sql: Relational data processing in spark." *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015

Spark SQL Optimizer Catalyst[1]

- All the statements are cached as Abstract Syntax Tree (AST)
- Lazy evaluation of AST enables optimization of expressed operations.
- Diagram here depicts optimization pipeline



[2] Armbrust, Michael, et al. "Spark sql: Relational data processing in spark." *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015

Spark SQL Optimizer Catalyst[1]

- Optimizer can work based on rules and use some cost base model for choosing a execution plan.
- Catalyst is designed to be Extensible, and workload specific optimizers can be created.

[2] Armbrust, Michael, et al. "Spark sql: Relational data processing in spark." *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015

References

[1] Armbrust, Michael, et al. "Spark SQL: Relational data processing in spark." *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015.

[2] Spark SQL Getting Started: https://spark.apache.org/docs/latest/sql-getting-started.html

[3] Spark SQL reference https://docs.databricks.com/spark/latest/spark-sql/index.html

[4] Documentation pages (version 2.4) https://spark.apache.org/docs/2.4.0/