trainologic

Spark 2.3

Spark SQL

trainologic

Spark SQL

'Spark SQL is a Spark module for structured data processing '

1. 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.
2. Internally, Spark SQL uses this extra information to perform extra optimizations.
3. There are several ways to interact with Spark SQL including SQL and the Dataset API.
4. 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.

→ **Taken from the official site: https://spark.apache.org/docs/latest/sql-programming-guide.html**

trainologic

DataFrame and Datasets

- Originally Spark provided only DataFrames.
- A DataFrame is conceptually a table with **typed columns**.
- However, it is **not typed at compilation**.
- Starting with Spark 1.6, Datasets were introduced.
- Represent a table with columns, but, the row is typed at compilation.

trainologic

Spark Session

- The entry-point to the Spark SQL.
- It doesn’t require a SparkContext (is managed within).
- Supports binding to the current thread (Inherited ThreadLocal) for use with getOrCreate().
- Supports configuration settings (SparkConf, app name and master).

```
SparkSession spark = SparkSession
      .builder()
      .appName("App Name")
      .config("spark.some.config.option", "some-value")
      .getOrCreate();
```

trainologic

Creating DataFrames

DataFrames are created using spark session

- ```
  Dataset<Row> revenueDF = spark.read().json("src/main/resources/sparksql/revenues.csv");
  ```
- ```
  Dataset<Row> revenueDF = spark.read().parquet("src/main/resources/sparksql/revenues.csv");
  ```

Options for constructing the DataFrame

- spark.read.option(“key”,”value”).option…read.csv(..):

  ```
  Dataset<Row> revenueDF = spark.read().option("header", true).csv("src/main/resources/sparksql/revenues.csv");
  ```

trainologic

Creating DataFrames

1. When reading files using built-in methods (spark.read.xxxx(path)) it'll be read as DataFrame (i.e. Dataset<Row>)
2. Spark will try to resolve the schema out of the file:
   a. csv - looking at sample of the values (you can't guess how 5 will be read…)
   b. parquet, avro - using intenal schema
3. Row - cause Spark can't guess beforehand what will be the types (kind of Java's Object)

trainologic

Schema

- Spark knows in many cases to infer the schema of the data by himself. (By using reflection or sampling)
- Use printSchema() to explore the schema that spark deduce
- Spark also enables to provide the schema yourself
- And to cast types to change existing schema – df.withColumn("new_col", df.col("col").cast(IntegerType))

trainologic

Exploring the DataFrame - Select, Filter

Select

- `revenueDF.select("product");`
- `revenueDF.select(col("product"), col("category"), col("revenue"));`

Filtering by columns

- `revenueDF.filter("revenue > 3000");`
- `revenueDF.filter(col("category").equalTo("Tablet"));`

trainologic

Exploring the DataFrame - columns

Drop

- `revenueDF.drop("category");`

withColumn

1. Static value

   ```
   Dataset<Row> wcDF = revenueDF.withColumn("ones", lit(1));
   ```

2. Based on other fields:

   ```
   revenueDF.withColumn("more_money", revenueDF.col("revenue").plus(3000));
   ```

3. Casting

   ```
   revenueDF .withColumn("int_revenue", revenueDF.col("revenue").cast(IntegerType));
   ```

trainologic

DataFrames, DataFrames...

trainologic

DataFrames, DataFrames...

1. Load ‘airports.csv’ file into Dataframe
2. Print the schema
3. Find all airports that located in ‘Greenland’
4. Find all airports that located in country started with ‘a’ AND the airport name is no less than 2 words.
5. Add column for the length of the airport name

trainologic

Exploring the DataFrame - Aggregations

```
revenueDF.groupBy("category").count();
```

Or

```
RelationalGroupedDataset rgs = wColDF.groupBy("category");
rgs.sum("revenue")
```

trainologic

Exploring the DataFrame - Aggregations

```
revenueDF.groupBy("category").count();
```

Or

```
RelationalGroupedDataset rgs = wColDF.groupBy("category");
rgs.sum("revenue")
```

Alias

```
revenueDF.withColumnRenamed("revenue", "revenue2")
```

trainologic

Exploring the DataFrame - Unions, Join

Union

1. Add rows to other rows
2. Needs the same schema

```
revenueDF.union(revenueDF)
```

Join

1. You should specify the columns to join on
2. You can specify the join type

```
revenueDF.join(revenueDF, "product");
```

trainologic

Aggregations...

trainologic

DataFrames, DataFrames...

```
.orderBy(desc("count"));
import static org.apache.spark.sql.functions.*;
```

1. Load ‘airports.csv’ and ‘flights.csv’ into Dataframe
2. Airports:
   a. How many different countries?
   b. How many airports per country?
   c. Printout the top 20 countries who has the bigest airport number?
3. And flights:
   a. How many flights per Country, DayofMonth there is?
   b. What was the busiest hour per country in 3.1.2008?

trainologic

SQL on DataFrame

Register DF as an SQL Table

- df.createOrReplaceTempView(“table1”)

Now we can use sql queries:

- spark.sql(“select count(*) from table1”)

trainologic

SQL on DataFrame - UDF

1. UDF - User Defined Function
2. There a build in function: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF
3. The ability to add code to our Spark SQL (and not only…)

trainologic

SQL on DataFrame - UDF

1. Create the function:

```
UDF1<Integer, Double> square = new UDF1<Integer, Double>() {
  public Double call(Integer integer) throws Exception {
      return Math.sqrt(integer);
  }};
```

2. Register:

```
spark.udf().register("sqr", square, DataTypes.DoubleType);
```

3. Use:

```
spark.sql("select category, sqr(revenue) as rev_sum ");
revenueDF.select(callUDF("sqr", col("revenue"))).show();
```

trainologic

Aggregations...

trainologic

DataFrames, DataFrames... - SQL

1. Load ‘airports.csv’ and ‘flights.csv’ into Dataframe
2. Airports:
   a. How many different countries?
   b. How many airports per country?
   c. Printout the top 20 countries who has the bigest airport number?
3. And flights:
   a. How many flights per Country, DayofMonth there is?
   b. What was the busiest hour per country in 3.1.2008?

trainologic

Datasets...

Datasets are coll!

- Typed - checked in completion time.
- You need a VERY good reason to use RDD or DataFrame instead of Dataset
- Datasets can created from classes
- You will need to specify encoder (where is the Scala’s case classes when needed :( )

trainologic

Datasets...

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

trainologic

Datasets...

```
Dataset<Integer> intDS =  spark.createDataset(Arrays.asList(1, 2, 3, 4, 5), Encoders.INT());
```

Now, this is safe:

```
intDS.filter((FilterFunction<Integer>) i -> i > 2);
```

Remember this (on dataframe)?

```
intDF.filter((FilterFunction<Row>) row -> row.getInt(0) > 3)
```

trainologic

Saving ~~Will~~ Datasets...

Spark provides usuefal method to save your dataset.

1. Each dataset has save method:
   ```
   revenueDataset.write().mode("overwrite").csv(path);
   ```
2. You can choose between few formats: csv, parquet, json, text and more
   → json is actually lined json, i.e. lines that each line is json.
3. The default is to write one file per partition, it can be changed:
   ```
   flightsDF.coalesce(3).write().mode("overwrite").json(path);
   ```

trainologic

Saving ~~Will~~ Datasets - Save Modes

Dataset provides 4 saving modes:

1. overwrite - overwrite file if exists
2. append - append to file
   a. Actually it added the new parts into the directory
   b. Overtime might yield in files explosion
   c. Harder to maintain reruns and atomic operations
3. ignore - ignore the operation
4. error, errorifexists, default - error if the file exists

   → This is the default mode

trainologic

Saving ~~Will~~ Datasets - Partitions

1. In the default status Spark will create one file per partition
2. coalesce - reduces the number of partitions

   ```
   flightsDF.coalesce(3)
   ```

   a. Not just for writing - good for optimize jobs (on that - later)
   b. No shuffling occurs - it just write more than one partition to one file
   c. *coalesce(1)* - can be good for using the outfile manually (e.g. csv file) but be careful with what you wish for!
3. repartition - column or just rempartion...

trainologic

A word about files type

1. csv - as poor performance and usability.
   Good for tests, debug and export data
2. Parquet - columnar storage format.
   Performed very well
3. If you don’t know what to use - use Parquet.
   Else → use Parquet ;)

trainologic

A word about compression

1. It is a good idea to compress your data (why?)
2. Snappy is widely used

trainologic

tf-idf

trainologic

tf-idf

Tf-idf - method in information retrieval to provide score to word - document

tf - term frequency
number of the term occurrences / number of terms in the document

idf - inverse document frequency
log_e(Total number of documents / Number of documents with term t in it)

trainologic

tf-idf

tf - term frequency
number of the term occurrences / number of terms in the document
idf - inverse document frequency
log_e(Total number of documents / Number of documents with term t in it)

TF

1. Create function the gets spark session and document and calculates the tf of the document
2. Create function that get spark session and list of documents and return on tf dataframe <document, word, tf>

trainologic

Web UI

- Spark provides rich web ui
- It can help to monitor and debug problems

trainologic

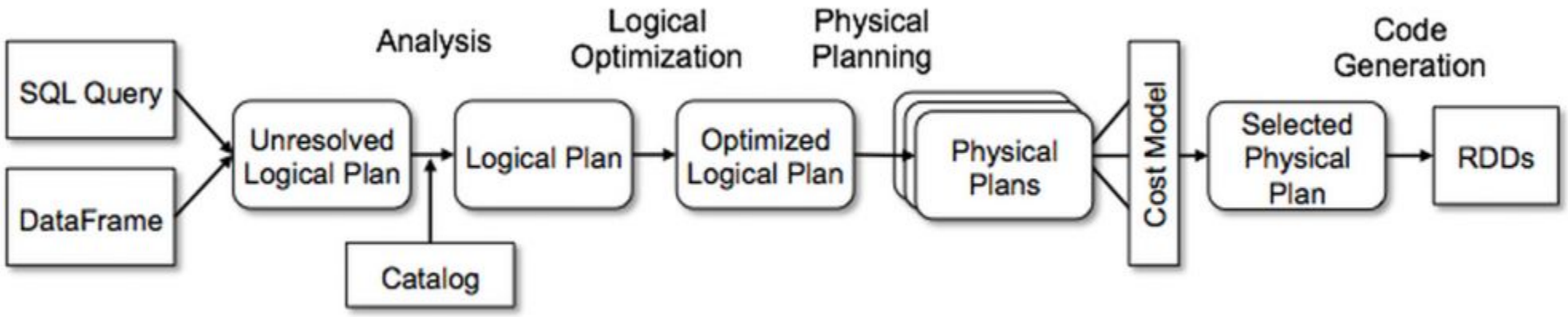
Query planning - Catalyst

- All of the Spark SQL APIs (SQL, Datasets and DataFrames) result in a Catalyst execution.
- Basicly Catalyst build an execution tree
- Trees can be created out of other trees using trasformations

trainologic

Query planning

- Query execution is composed of 4 stages:
  - Logical plan.
  - Optimized logical plan.
  - Physical plan.
  - Code-gen (Tungesten).



trainologic

Logical plan

- Sometimes called Analysis
- In the first stage, the Catalyst creates a logical execution plan.
- A tree composed of query primitives and combinators that
- match the requested query.
- Mapping names to value with unique ids
- Optimzies expressions like col = col
- And much more...

trainologic

Optimized logical plan

- Catalyst runs a set of rules for optimizing the logical plan.
- Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs.
- E.g.:
  - Merge projections.
  - Push down filters.
  - Convert outer-joins to inner-joins.
  - Cast types

trainologic

Physical Planning

- Spark SQL takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine
- Than it selects the best plan
- e.g. joins - if the dataset is small it'll use broadcast join
- rule-based physical optimizations, such as pipelining projections or filters into one Spark map operation
- it can push operations from the logical plan into data sources that support predicate or projection pushdown

trainologic

Tungesten

- Spark 2 introduced a new optimization: Whole stage code generation.
- It appears that in many scenarios, the bottleneck is the CPU.
- Wasted on virtual calls and cache-line swapping.
- Simple and query specific code should be favored.

trainologic

Tungesten

Tungesten:

1. Memory Management and Binary Processing:
   leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
2. Cache-aware computation:
   algorithms and data structures to exploit memory hierarchy
3. Code generation:
   using code generation to exploit modern compilers and CPUs

trainologic

Query planning

- All of the Spark SQL APIs (SQL, Datasets and DataFrames) result in a Catalyst execution.
- Query execution is composed of 4 stages:
  - Logical plan.
  - Optimized logical plan.
  - Physical plan.
  - Code-gen (Tungesten).

trainologic

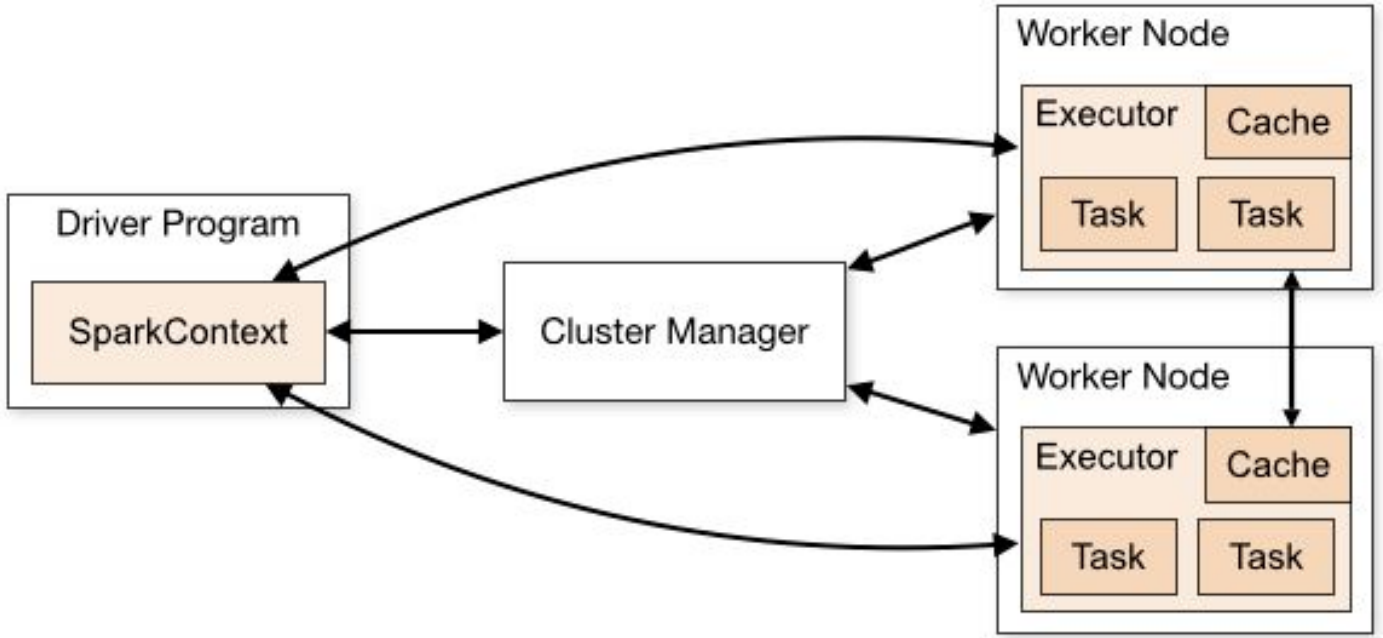
Integrating Spark with your App

- How does it work?
- Server vs. client mode

trainologic

Integrating Spark with your App

Remember we talked about this



trainologic

How does it work?

- Cluster manager
  - Responsible for the resource allocation
  - Cluster management
  - Orchestrates work
- Worker nodes:
  - Tracks and spawn application processes

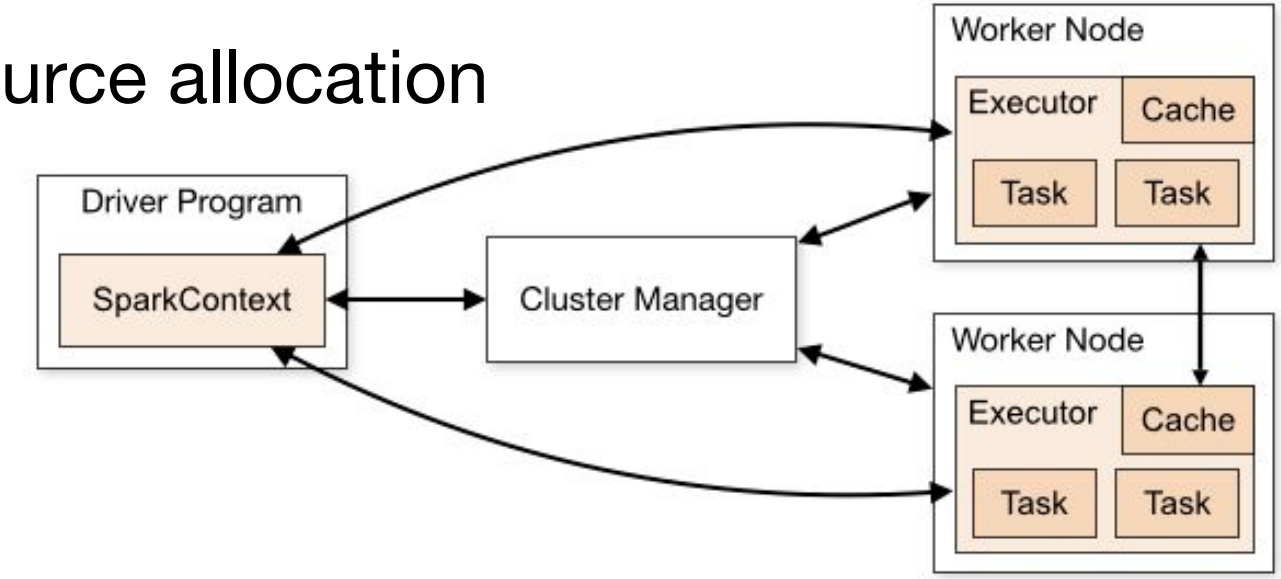


Image taken from https://spark.apache.org/docs/latest/cluster-overview.html

trainologic

How does it work?

- Driver:
  - Executes the main program
  - Creates the RDDs
  - Collects the results
- Executors:
  - Executes the RDD operations
  - Participate in the shuffle

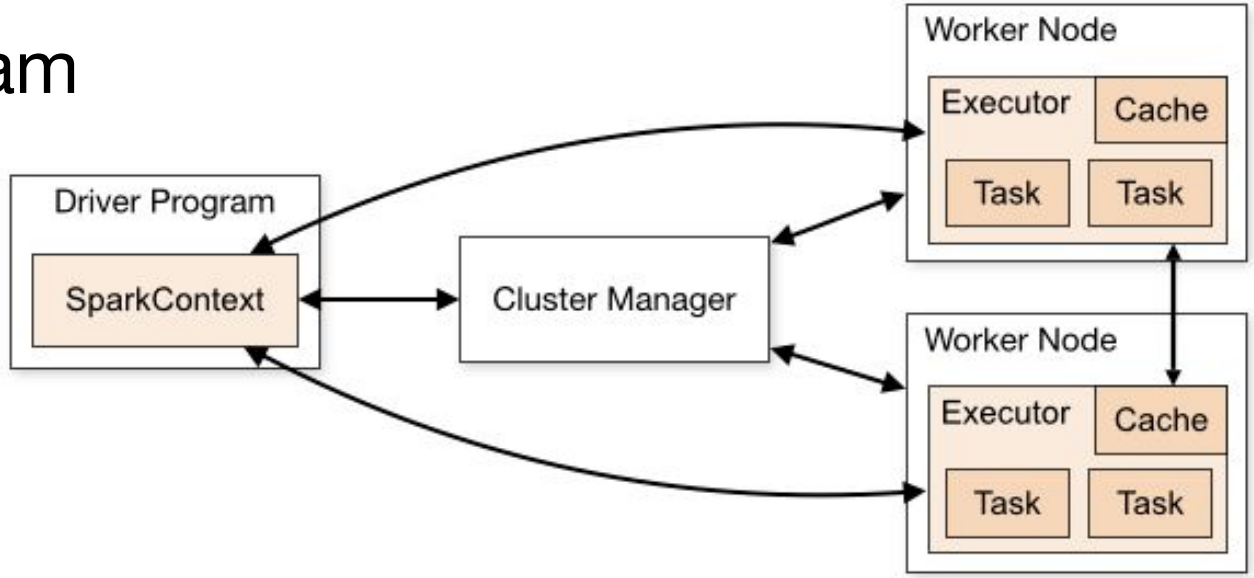


Image taken from https://spark.apache.org/docs/latest/cluster-overview.html

trainologic

Spark Cluster Manager

- Stand alone
- Mesos
- Yarn
- Kuberntics
- EMR/Azure

trainologic

Spark Client Mode

- The driver is in your app
- Easy to retrieve results
- If the app crashes – spark crashes
- *--deploy-mode client*

trainologic

Spark Cluster Mode

- The driver is running on the worker
- The app can recover from crash (using *--supervise* flag)
- Recommended when the launching app is far from the cluster
- On Yarn - get one more container to run on
- Harder to communicate with
- *--deploy-mode cluster*

trainologic

Spark Submit

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  examples/jars/spark-examples_2.11-2.2.0.jar \
  100
```

trainologic

How many Cores?

1. num-executors - the number of excutor
2. executor-cores - how many cores per executor
3. executor-memory - how manu memory per executor
4. num-executors * executor-cores == worker num
5. --driver-memory and --driver-cores - driver configuration

trainologic

How many Cores?

Let’s assume that we have 3 nodes, 16 cores. 64GB each

Option I

num-executors=3, executor-cores=16, executor-memory=64GB

→ We didn’t leave any room for operations system and the node manager...

trainologic

How many Cores?

Let’s assume that we have 3 nodes, 16 cores. 64GB each

Option II

num-executors=3, executor-cores=14, executor-memory=64GB

→ As cores using threads, this is too many threads for process

trainologic

How many Cores?

Let’s assume that we have 3 nodes, 16 cores. 64GB each

Option III

num-executors=45, executor-cores=1, executor-memory=64GB

→ Too many processes, e.g. broadcast variables will need to create again and again

trainologic

How many Cores?

Let’s assume that we have 3 nodes, 16 cores. 64GB each

Option IV

num-executors=9, executor-cores=5, executor-memory=20GB

1. 3 executors per node, 5 cores - total of 45 workers
2. 20GB per executors - about 4GB per worker

trainologic

How many Cores?

1. Use as least executors as possible
2. Use 4-7 cores per executor.
3. Secure at least 2GB per worker
4. Don’t exceed 64GB memory per executor
5. Don’t forget to leave 1-2 cores for the operation system and for the cluster manager