Spark Partitions, Coalesce and Repartition

PARTITIONS

- **Partitions are determined when files are read**
  - Core Spark determines RDD partitioning based on location, number, and size of files
    - Usually each file is loaded into a single partition
    - Very large files are split across multiple partitions
  - Catalyst optimizer manages partitioning of RDDs that implement DataFrames and Datasets

- partitions() Get the array of partitions of this RDD, taking into account whether the RDD is checkpointed or not
- getNumPartitions() - Returns the number of partitions of this RDD.

COALESCE AND REPARTITION

- The repartition algorithm does a full shuffle and creates new partitions with data that's distributed evenly. Let's create a DataFrame with the numbers from 1 to 12.
- coalesce uses existing partitions to minimize the amount of data that's shuffled. repartition creates new partitions and does a full shuffle. coalesce results in partitions with different amounts of data (sometimes partitions that have much different sizes) and repartition results in roughly equal sized partitions.
- **Is coalesce or repartition faster?**
- coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions. You'll usually need to repartition datasets after filtering a large data set. I've found repartition to be faster overall because Spark is built to work with equal sized partitions.
- repartition - it's recommended to use it while increasing the number of partitions, because it involve shuffling of all the data.
- coalesce - it's is recommended to use it while reducing the number of partitions. For example if you have 3 partitions and you want to reduce it to 2, coalesce will move the 3rd partition data to partition 1 and 2. Partition 1 and 2 will remains in the same container. On the other hand, repartition will shuffle data in all the partitions, therefore the network usage between the executors will be high and it will impacts the performance.
- **coalesce performs better than repartition while reducing the number of partitions.**

Spark Dataframes

Dataframe

DataFrames introduced in Spark 1.3 as extension to RDDs

Distributed collection of data organized into named columns

» Equivalent to Pandas and R DataFrame, but distributed

Types of columns inferred from values

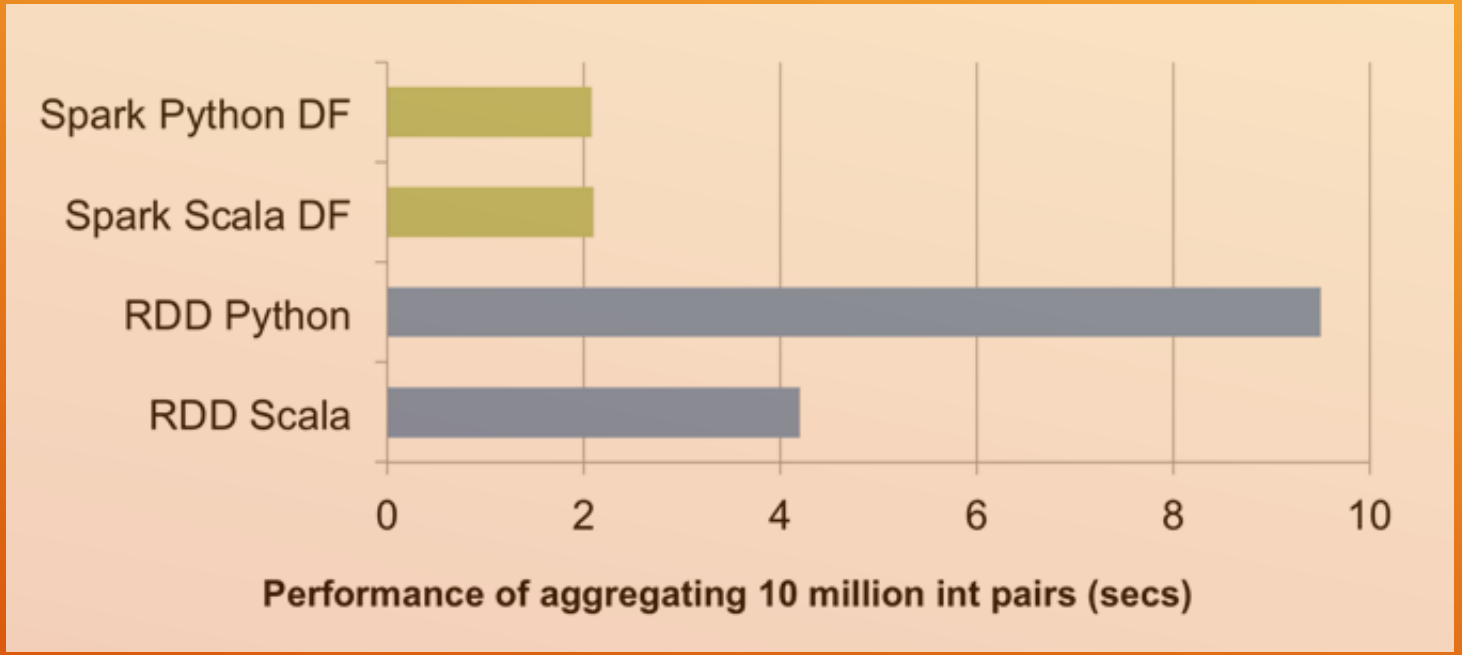
Easy to convert between Pandas and pySpark

» Note: pandas DataFrame must fit in driver

```
#Convert Spark DataFrame to Pandas
pandas_df =  spark_df.toPandas()
# Create a Spark DataFrame from Pandas
spark_df = sc.createDataFrame(pandas_df)
```

Dataframe

Almost 5x pySpark performance on a single machine



CREATING DATAFRAMES

- Using an RDD spark.createDataFrame(some_rdd)

and then passing rdd.toDF(List of Columns)

- Using a RDD created using Row type. and then converting it to Dataframe. here there is no need to use toDF

SCHEMA- STRUCTURE OF DATA

- 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).
- A schema is described using StructType which is a collection of StructField objects (that in turn are tuples of names, types, nullability classifier and metadata of key-value pairs).
- We can create Dataframe using StructType as well like the below

createDataFrame(another_rdd, schemaofstructtype)

STRUCT TYPE

- StructType and StructField belong to the org.apache.spark.sql.types package import org.apache.spark.sql.types.StructType

  schemaUntyped = new StructType()

  .add("a", "int")

  .add("b", "string")

- You can use the canonical string representation of SQL types to describe the types in a schema (that is inherently untyped at compile type) or use type-safe types from the org.apache.spark.sql.types package.

- // it is equivalent to the above expression

- import org.apache.spark.sql.types.{IntegerType, StringType}

schemaTyped = new StructType() .add("a", IntegerType) .add("b", StringType)

READING DATA FROM EXTERNAL DATA SOURCE

- You can create DataFrames by loading data from structured files (JSON, Parquet, CSV,ORC), RDDs, tables in Hive, or external databases (JDBC) using SQLContext.read method. read returns a DataFrameReader instance.
- Among the supported structured data (file) formats are (consult Specifying Data Format (format method) for DataFrameReader):
  - JSON
  - parquet
  - JDBC
  - ORC
- reader = spark.read.parquet("/path/to/file.parquet")

For **xml** –

```
Xml_df = spark..read.format("com.databricks.spark.xml") \
    .option("rootTag", "hierarchy") \
    .option("rowTag", "att") \
    .load("test.xml“)
```

For **Excel-**

https://forums.databricks.com/questions/18295/how-to-read-excel-file-using-databricks.html

QUERY and Processing DATAFRAME

Commonly used transformations are as below:

- select
- filter
- groupBy
- union
- explode
- where clause
- withColumn
- partitionBy
- orderBy
- rank etc

Writing Dataframes

- We can write the dataframes back to disk as files using any of the supported read formats using below syntax
- dataframe.write.<format>(“/location”)
- df.write.mode(SaveMode.Overwrite).format("<format>").save("/target/path”)

Schema Evolution

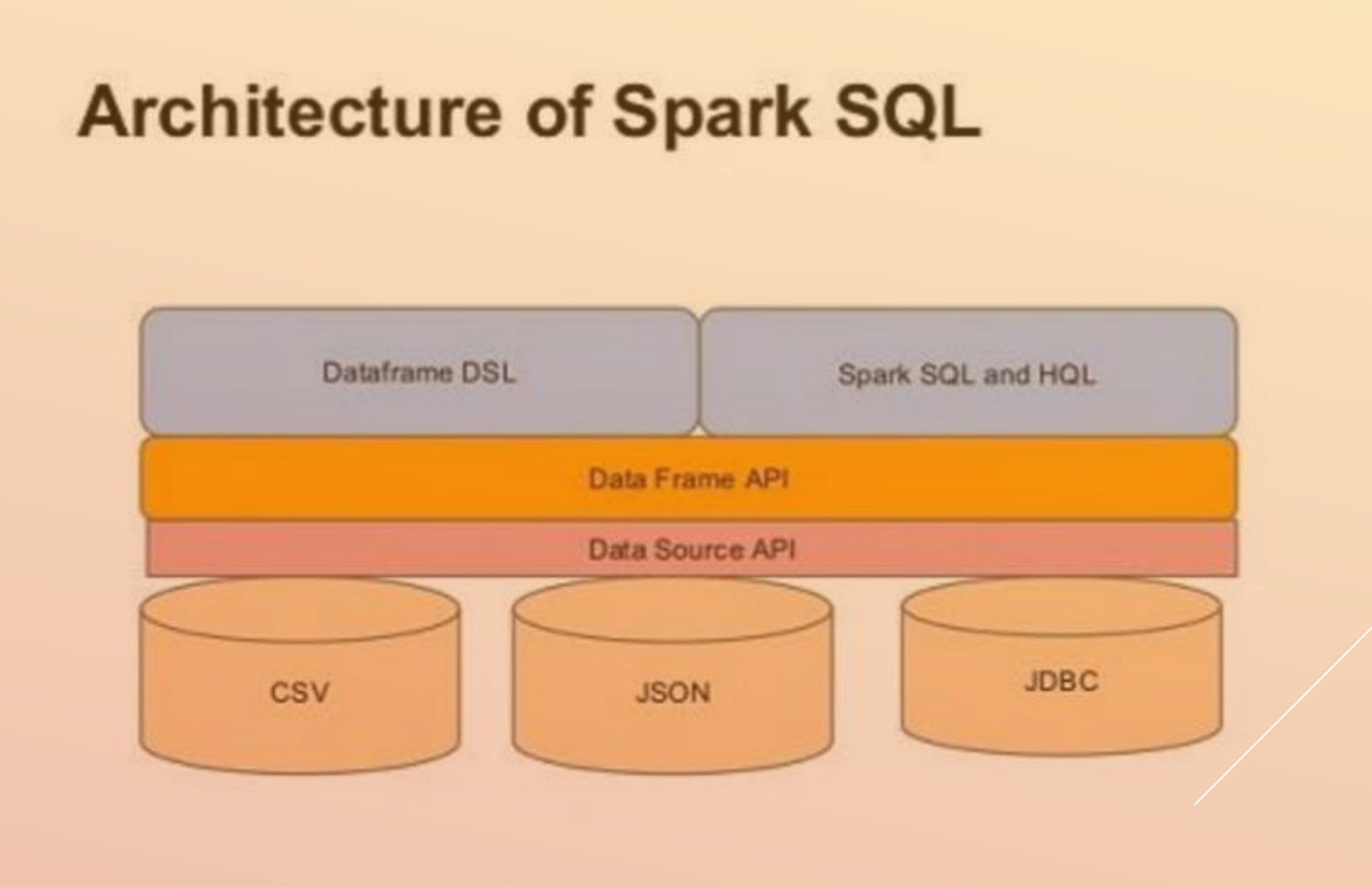
```
df = spark.read.option("mergeSchema","true").parquet(“/path")
```

Spark SQL

Why Spark SQL?

- Spark SQL originated as Apache Hive to run on top of Spark and is now integrated with the Spark stack. Spark SQL was built to overcome these drawbacks and replace Apache Hive.
- Limitations with Hive:
  - Hive launches MapReduce jobs internally for executing the ad-hoc queries. MapReduce lags in the performance when it comes to the analysis of medium sized datasets (10 to 200 GB).
  - Hive has no resume capability. This means that if the processing dies in the middle of a workflow, you cannot resume from where it got stuck.
  - Hive cannot drop encrypted databases in cascade when trash is enabled and leads to an execution error. To overcome this, users have to use Purge option to skip trash instead of drop.

Architecture of Spark SQL



SQL - A language for Relational DBs

SQL = Structured Query Language

Supported by pySpark DataFrames (SparkSQL)

Some of the functionality SQL provides:

» Create, modify, delete relations

» Add, modify, remove tuples

» Specify queries to find tuples matching criteria

●Register a DataFrame as a named temporary table to run SQL.

●df.registerTempTable("auctions") (1)

●spark.sql("SELECT count(*) AS count FROM auctions")

sql: org.apache.spark.sql.DataFrame = [count: bigint]

Queries in SQL

Single-table queries are straightforward

To find just names and logins:

```
SELECT * FROM Students S
WHERE S.age=18
SELECT S.name,S.login FROM Students S WHERE S.age=18
```

SparkSQL and Spark DataFrames join() supports:

» inner, outer, left outer, right outer, semijoin

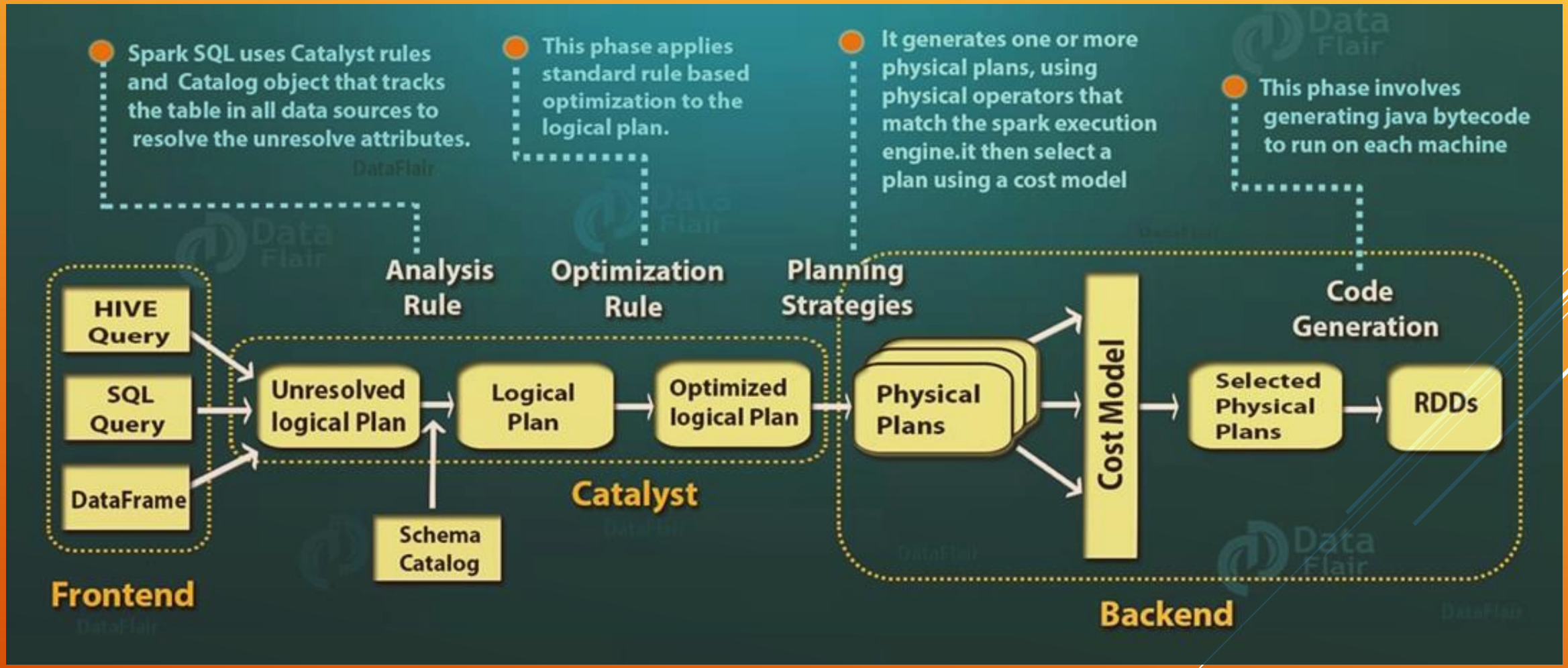
Spark Dataframes - Catalyst & Tungsten Optimizers

Catalyst Optimizer

Catalyst can improve SQL, DataFrame, and Dataset query performance by optimizing the DAG to

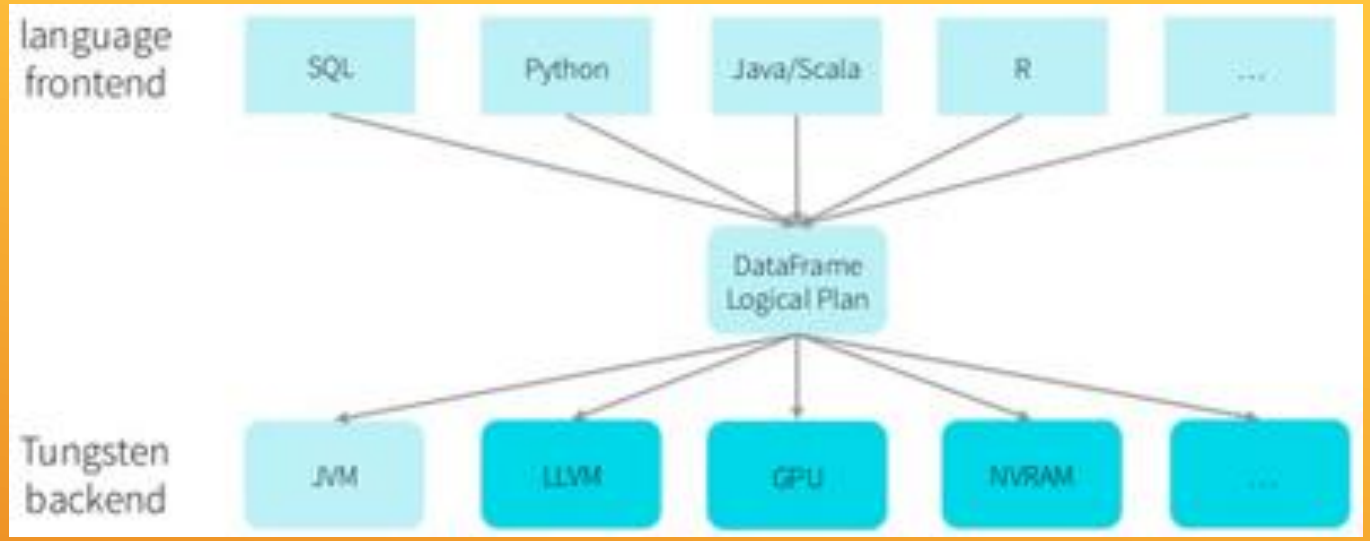
- Minimize data transfer between executors
  - Such as *broadcast* joins—small data sets are pushed to the executors where the larger data sets reside
- Minimize wide (shuffle) operations
  - Such as unioning two RDDs—grouping, sorting, and joining do not require shuffling
- Pipeline as many operations into a single stage as possible
- Generate code for a whole stage at run time
- Break a query job into multiple jobs, executed in a series

Catalyst Optimizer



https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

Project Tungsten



Optimization Features

- **Off-Heap Memory Management** - using binary in-memory data representation aka Tungsten row format and managing memory explicitly,
- **Cache Locality** - cache-aware computations with cache-aware layout for high cache hit rates,
- **Whole-Stage Code Generation** (aka *CodeGen*).-improves the execution performance of a query by collapsing a query tree into a single optimized function that eliminates virtual function calls and leverages CPU registers for intermediate data.

https://community.cloudera.com/t5/Community-Articles/What-is-Tungsten-for-Apache-Spark/ta-p/248445

RDD vs Dataframe vs Dataset

RDD vs DataFrame vs Dataset

When to use RDD:

- 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.

RDD vs DataFrame vs Dataset

DataFrames

Like an RDD, a DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction

Datasets

Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a *strongly-typed* API and an *untyped* API. Conceptually, consider DataFrame as an *alias* for a collection of generic objects *Dataset[Row]*, where a *Row* is a generic *untyped* JVM object. Dataset, by contrast, is a collection of *strongly-typed* JVM objects, dictated by a case class you define in Scala or a class in Java.

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

Thank you!