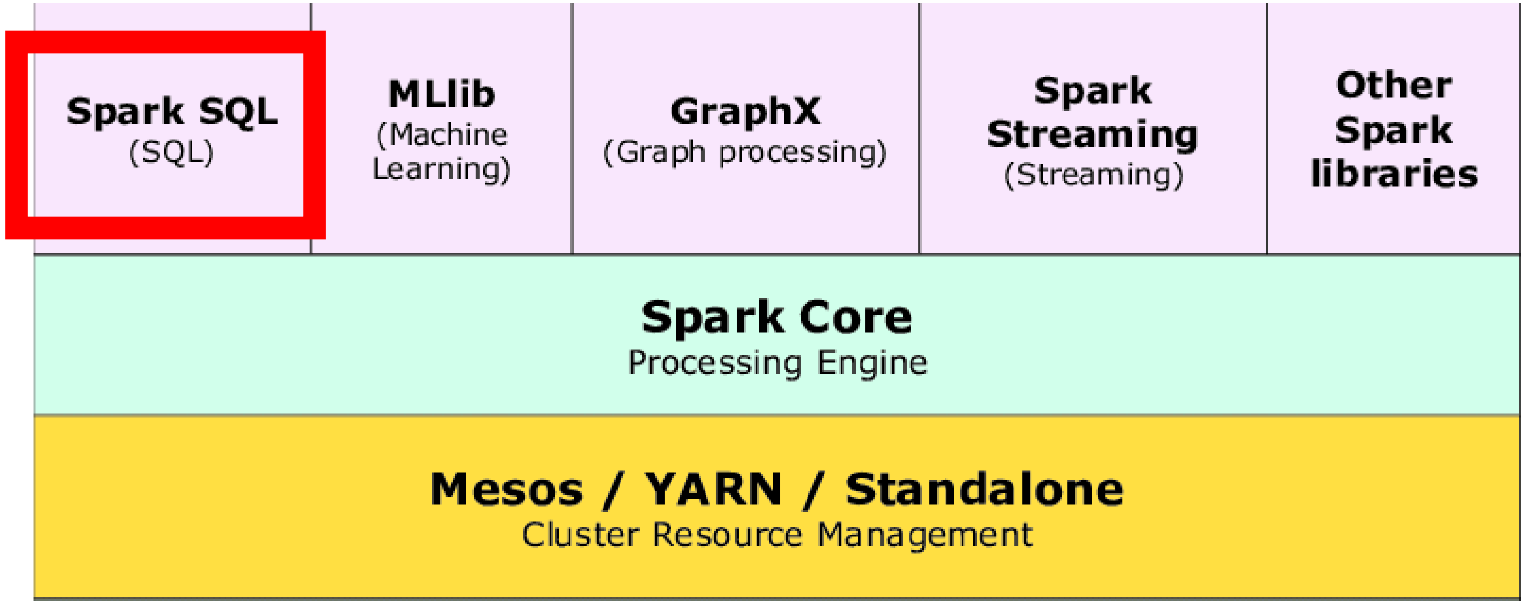
BIG DATA PROCESSING

EGCI 466

Spark SQL

Spark Stack



- Querying data
- Streaming
- Machine Learning Library
- Graph

Use RDD for programming abstraction (Resilient Data Distribution)

Carry data across many computing nodes in parallel, and transform it.

SPARK SQL

- Import relational data
- Enables queriying structured and unstructured data
- Provide a common query language
- Provide APIs for Scala, Java and Python --> RDDs

Dataframes

- Distributed collection into data and header
- Similar to data frame in Pandas(python) but with richer optimizations
- Build on top of RDDS
- It can performs relatonal queries

DataFrame Benefits

- Ability to scale from kilobytes of data → petabytes on a large cluster
- Support for a wide array of data formats and storage
- State-of-the art optimizer/code generator through Catalyst Optimizer
- Integrations with many tools/ API for many languages

Initialization

```
# Creating a spark context class
sc = SparkContext()

# Creating a spark session
spark = SparkSession \
     .builder \
     .appName("Python Spark DataFrames basic example") \
     .config("spark.some.config.option", "some-value") \
     .getOrCreate()
```

Create/ Import data using python

```
if 'spark' in locals() and isinstance(spark, SparkSession):
    print("SparkSession is active and ready to use.")
else:
    print("SparkSession is not active. Please create a SparkSession.")
```

```
df =sqlContext.read.json(“filename”);
df.show()
df.printSchema()
```

Example Data Frame

Input JSON file

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

Created DataFrame

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

Create Schema

```
pyspark.sql.types import
StructType,StructField,StringType,FloatType
schema=StructType([ \
      StructField("fistName",StringType(),True),\
      StructField("lastName",StringType(),True),\
      StructField("Grade",Floatype(),True),\
      ])

df=sc.createDataFrame(data=data2,schema=schema)
```

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

Simple Dataframes operations

- Select operation
  - df.select([‘age’,name’]).show()

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

  - df.select(df["name"]).show()

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

Mahidol University
Wisdom of the Land

Create a view

```
df.createTempView("people")
df.createOrReplaceTempView("people")
```

SQL Query

```
spark.sql("SELECT age, name
FROM people WHERE age >
21").show()
```

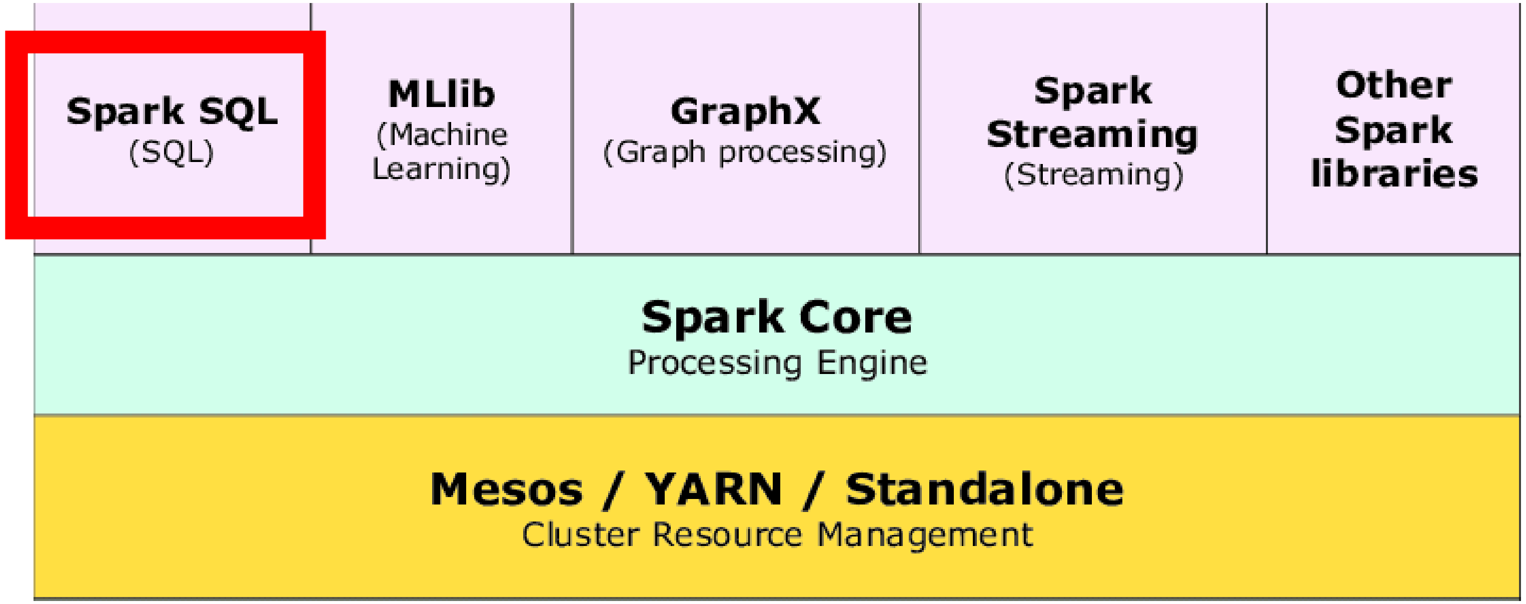
DataFrame Python API

```
df.filter(df["age"]>21).show()
```

Result

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

Spark Stack



- Querying data
- Streaming
- Machine Learning Library
- Graph

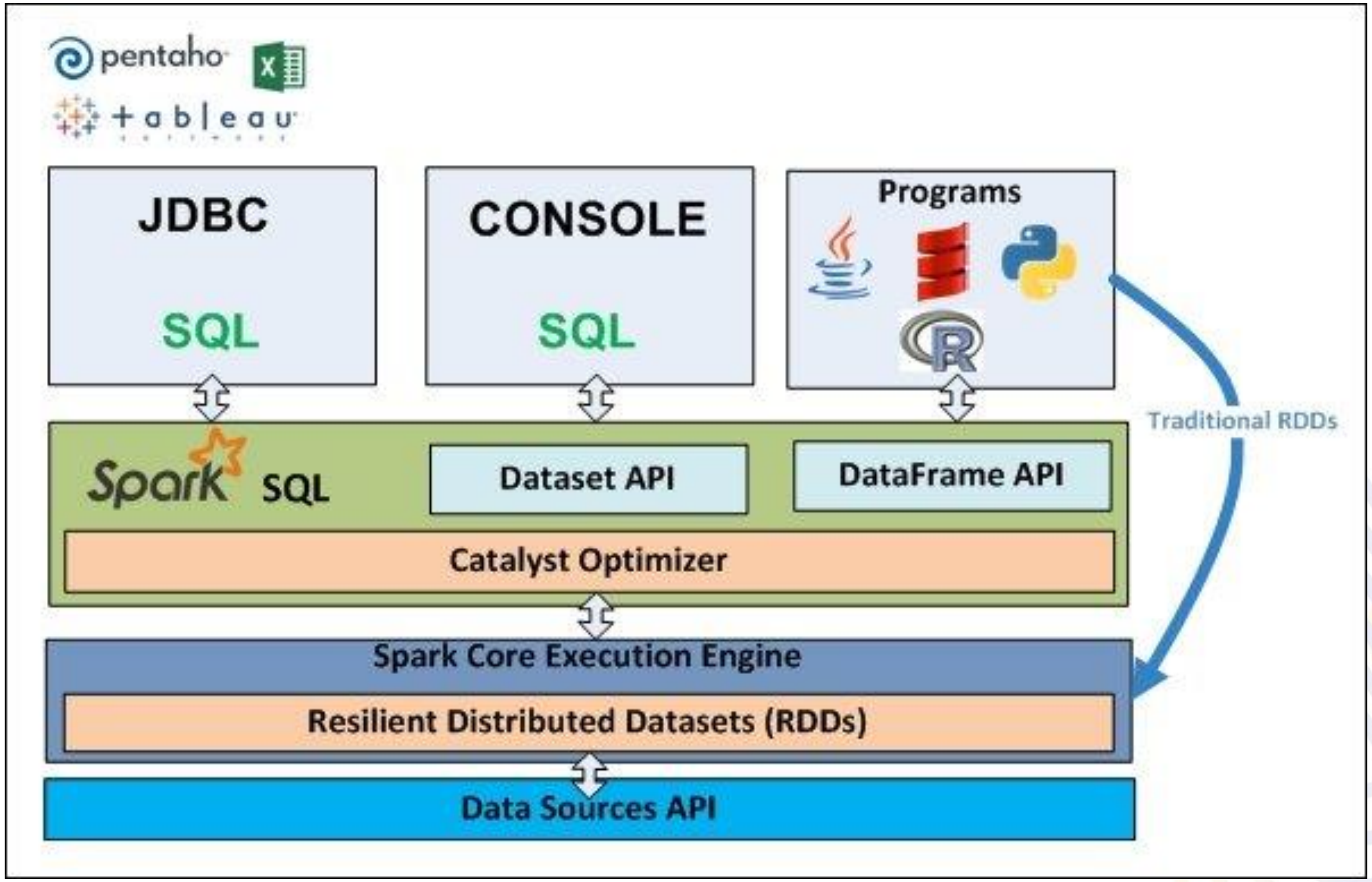
Use RDD for programming abstraction (Resilient Data Distribution)

Carry data across many computing nodes in parallel, and transform it.

SPARK SQL

- Import relational data
- Enables queriying structured and unstructured data
- Provide a common query language
- Provide APIs for Scala, Java and Python --> RDDs

SPARK SQL


pentaho
tableau
JDBC
SQL
CONSOLE
SQL
Programs
Traditional RDDs
Spark SQL
Dataset API
DataFrame API
Catalyst Optimizer
Spark Core Execution Engine
Resilient Distributed Datasets (RDDs)
Data Sources API

Create/ Import data using python

```
pyspark.sql sqlcontext
sqlContext = SQLContext(sc)

df =sqlContext.read.json("filename");
df.show()
```

SPARK SQL

- Spart module for structured data processing
- Import relational data
- Enables queriying structured and unstructured data
- Provide a common query language
- Provide APIs for Scala, Java and Python --> from imported RDDs

SPARK SQL - Benefits

- Includes a cost-based optimizer, columnar storage, and code generation to make queries fasts
- Scales to thousands of nodes and multi-hour queries using Spark engines(provides full mid-query fault tolerance)
- Provide programming abstraction: DataFrames
- Act as distributed query engine

DATAFRAMEs

- Distributed collection of data organized into named column
- Conceptualy equivalent to a table in a relational database or a data frame in R/Python but with richer optimization
- On top of RDD API

Create/ Import data using python

```
pyspark.sql sqlcontext
sqlContext = SQLContext(sc)

df =sqlContext.read.json("filename");
df.show()
df.printSchema()

#Register the DataFrame as a SQL temporary view
df.crateTempView("People")
```

Example Data Frame

Input JSON file

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

Created DataFrame

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

DataFrame Benefits

- Ability to scale from kilobytes of data → petabytes on a large cluster
- Support for a wide array of data formats and storage
- State-of-the art optimizer/code generator through Catalyst Optimizer
- Integrations with many tools/ API for many languages

Example

SQL Query

```
spark.sql("SELECT age, name
FROM people WHERE age >
21").show()
```

DataFrame Python API

```
df.filter(df["age"]>21).show()
```

Result

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

Create Schema

```
pyspark.sql.types import StructType,StructField,StringType,FloatType
schema=StructType([ \
        StructField("fistName",StringType(),True),\
        StructField("lastName",StringType(),True),\
        StructField("Grade",Floatype(),True),\
        ])

df=sc.createDataFrame(data=data2,schema=schema)
```

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

DataFrame Operatiohns

- Rename Column
  - df.withColumnRenamed('name','Students_name').show()
- Select operation
  - df.select(['name','gpa']).show()
  - df.select(df["name"]).show()
  - spark.sql("SELECT name FROM people").show()
- Sort operation
  - df.sort('age').show()
  - spark.sql("SELECT * FROM people order by age").show()

Create DataFrame

```
from pyspark.sql import SparkSession
```

- ```
  spark = SparkSession \
          .builder \
          .appName("Create_df") \
           .getOrCreate()
  ```

- Create from RDD
  - `dfFromRDD=rdd.toDF()`
  ```

Create Schema

```
pyspark.sql.types import
StructType,StructField,StringType,FloatType
schema=StructType([ \
      StructField("fistName",StringType(),True),\
      StructField("lastName",StringType(),True),\
      StructField("Grade",Floatype(),True),\
      ])

df=sc.createDataFrame(data=data2,schema=schema)
```

https://spark.apache.org/docs/latest/sql-ref-datatypes.html

Import DataFrame

```
pyspark.sql sqlcontext
sqlContext = SQLContext(sc)
```

```
df =sqlContext.read.json("filename");
df.show()
```

- Make SQL temporaly view
  - `df.createTempView("people")`
  - `df.createOrReplaceTempView("people")`

DataFrame Operations

- Rename Column
  - `df.withColumnRenamed('name','Students_name').show()`
- Select operation
  - `df.select(['name','gpa']).show()`
  - `df.select(df["name"]).show()`
  - `spark.sql("SELECT name FROM people").show()`
- Sort operation
  - `df.sort('age').show()`
  - `spark.sql("SELECT * FROM people order by age").show()`

DataFrame operations

- Create a new column
  - `df_new=df.withColumn("Graduation age",df["age"]+1)`
  - `df_new.show()`

- Create a new column
  - `df_new.drop('Graduation age','gpa')`
  - `df_new.show()`

Filter /Group by

- Simple Filter
  - `df.filter(df["age"] > 21)`
  - `spark.sql("SELECT * FROM people WHERE age > 21")`

  - AND operation
  - `df["age"] > 21 & df[“department”]==‘xxx’`

Grouping and aggregation

- Average
  - `df.groupby(“department”).agg({”age": "AVG"})`
  - `spark.sql("SELECT avg(age) FROM people group by department)`
- Count
  - `spark.sql("SELECT count(age) FROM people group by department order by count(age) desc”)`
  - `df.groupby(“department”).agg({”age": ”count"}) .sort("count(age)", ascending=False`
- **Join**
  - `Combined_df= df1.join(df2, on=“id”,how=“inner”)`

Simple Pre-processing with NULL

- Drop NULL VALUE
  - `df_new=df.na.drop()# Remove all rows with NULL`
  - `df_new=df.na.drop(thresh=3) # Remove all rows with 3 non-NULL`
  - `df_new=df.na.drop(how=all) # Remove rows with all NULL`

- Fill NULL VALUE (from pyspark.sql.functions import mean)
  - `df.na.fill(df.select(mean(df['b'])).collect()[0][0],['b'])`

Distinct/Drop duplication

- Distinct
  - `df.distinct.show()`
  - `df.select([‘age’)).distinct().show()`
  - `spark.sql("SELECT distinct age FROM people)`
- Drop Duplication
  - `df.dropDuplicates()`
  - `df.select([‘age’)).distinct().show()`

User Defined Function (UDF)

- Simple funciton
  - def UpperCase(str):
    return str.upper()
- Create UDF
  - `upperCaseUDF=udf(lambda z: UpperCase(z))`
- Apply UDF
  - `df.withColumn(“Uppercase Name”,upperCaseUDF(“Name”))`

Dataset

- Distributed collection of data
  - Consists of a collection of **strongly typed** JVM objects (Explicit during creation)
  - Provides the combined benefits of both RDDs and Spark SQL

Datasets Features

- Immutable: cannot be deleted or lost
- Feature an encoder that converts JVM objects to a tabular representation
- Extend DataFrame Type-safe and object-oriented API capabilities
- Work with both Scala and Java APIs

Datasets Benefit

- Compile-time type safety
- Compute faster than RDDs
- Offer benefits of SparkSQL and DataFrame
- Optimize queries using Catalyst and Tungsten
- Enable improved memory usage and chaching

Dataset example operation

- val ds=Seq(“Alpha”,”Beta”,”Delta”,”Omicron”).**toDS()**
- val ds1=spark.read.text(“file.txt”).**as[String]**
- ds.map(**lambda:…**).collect()

https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#creating-datasets

https://www.educba.com/spark-dataset/

Datasets

RDDs vs Dataframes vs Datasets

	RDDs	Dataframes	Datasets
Data Representation	RDD is a distributed collection of data elements without any schema.	It is also the distributed collection organized into the named columns (Not typesafe)	It is an extension of Dataframes with more features like type-safety and object-oriented interface.
Optimization	No in-built optimization engine for RDDs. Developers need to write the optimized code themselves.	It uses a Catalyst and Tungsten for optimization.	It also uses a Catalyst and Tungsten for optimization purposes.
Projection of Schema	Here, we need to define the schema manually.	It will automatically find out the schema of the dataset.	It will also automatically find out the schema of the dataset by using the SQL Engine.
Aggregation Operation	RDD is slower than both Dataframes and Datasets to perform simple operations like grouping the data.	It provides an easy API to perform aggregation operations. It performs aggregation faster than both RDDs and Datasets.	Dataset is faster than RDDs but a bit slower than Dataframes.

https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets/

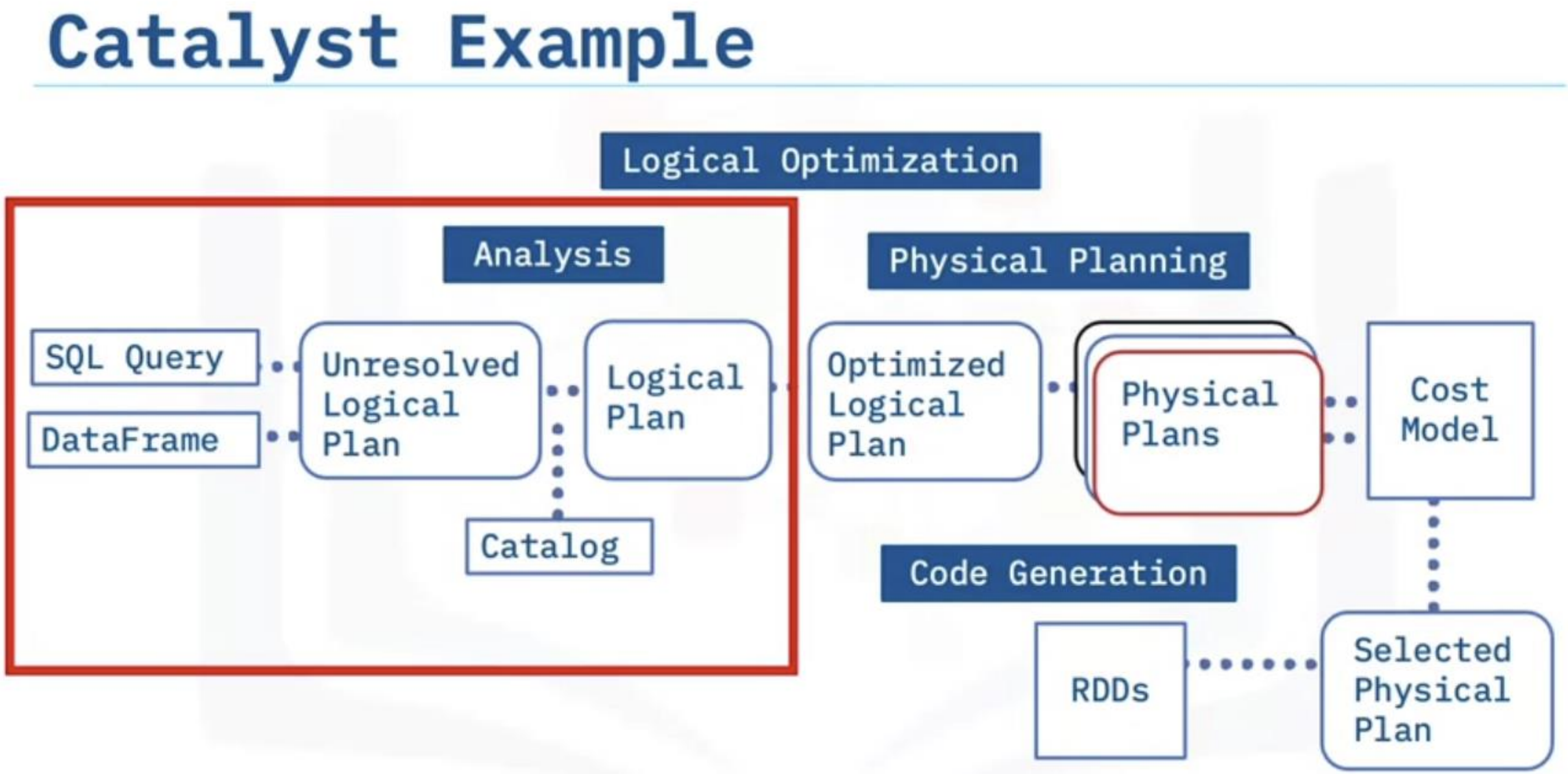
Memory optimization

- Reduce query time
- Reduce memory
- Reduce cost

Optimizer: Catalyst

- ***Rule-based*** optimizer
  - define how to run the query
  - Indexing
  - query contain only requried column
- Use a tree data structure and a set of rules
- Add new optimization easily
- Enable developer to add data source-specific rules and support new data types

Steps of catalyst



Catalyst Phases

Analysis phase

Analyze the query, the DataFrame to create a logical plan.

Logical Optimization phase

Create the rule-based optimization step of Spark SQLDuring the **Physical Planning phase**,

Generates multiple physical plans and chose the plan with the least computational cost

Code Generation.

Applies the selected physical plan and generates Java bytecode to run on each node.

Tungsten:optimizer

- *cost-based* optimizer
  - Based on CPU/ memory performance
  - Not I/O
- Manages memory explicitly
- Creates cache-friendly data structures that are arranged easily and more securely
- Use STRIDE-based memory access instead of (RAM)
- Does not enable virtual function dispatches, reducing multiple CPU calls.
- Tungsten places intermediate data in CPU registers and enables loop unrolling.

Catalyst and Tungsten optimizer summary

- Catalyst is the Spark SQL built-in rule-based query optimizer.
- Catalyst performs analysis, logical optimization, physical planning, and code generation.
- Tungsten is the Spark built-in cost-based optimizer for CPU and memory usage that enables cache-friendly computation of algorithms and data structures.

Using optimized results

```
# Enable adaptive query execution
    spark.conf.set("spark.sql.adaptive.enabled", "true")
# Applying Adaptive Query Execution (Runtime adaptive optimization)
    optimized_join = df1.join(df2, on="name")
# Show the optimized join result
    print("Optimized Join DataFrame:")
    optimized_join.show()
```

Assignments: Integration

- Use All of the following input files
  - Words.txt
  - Alice.txt
  - Covid.txt
- Run a wordcount (remove all punctuations)
- Transform the output to dataframe (header is **word** and **count**)
- Use sql to help lowercase all and filter on the one with wordcount more than 10
- **Join** all output as a data frame
- Save the output

Want to run using command line??: https://www.cloudskillsboost.google/catalog_lab/719