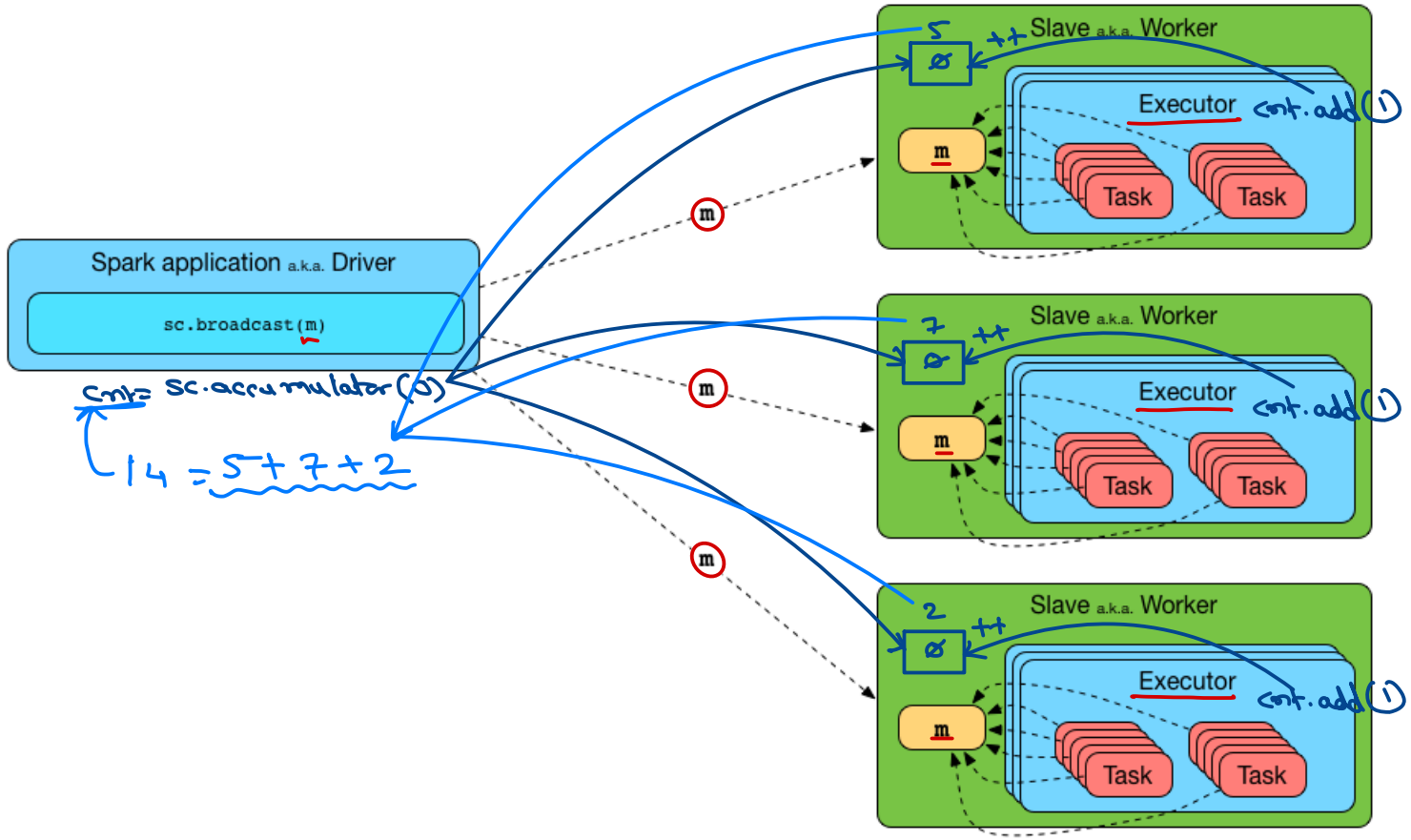
Apache Spark

Sunbeam Infotech

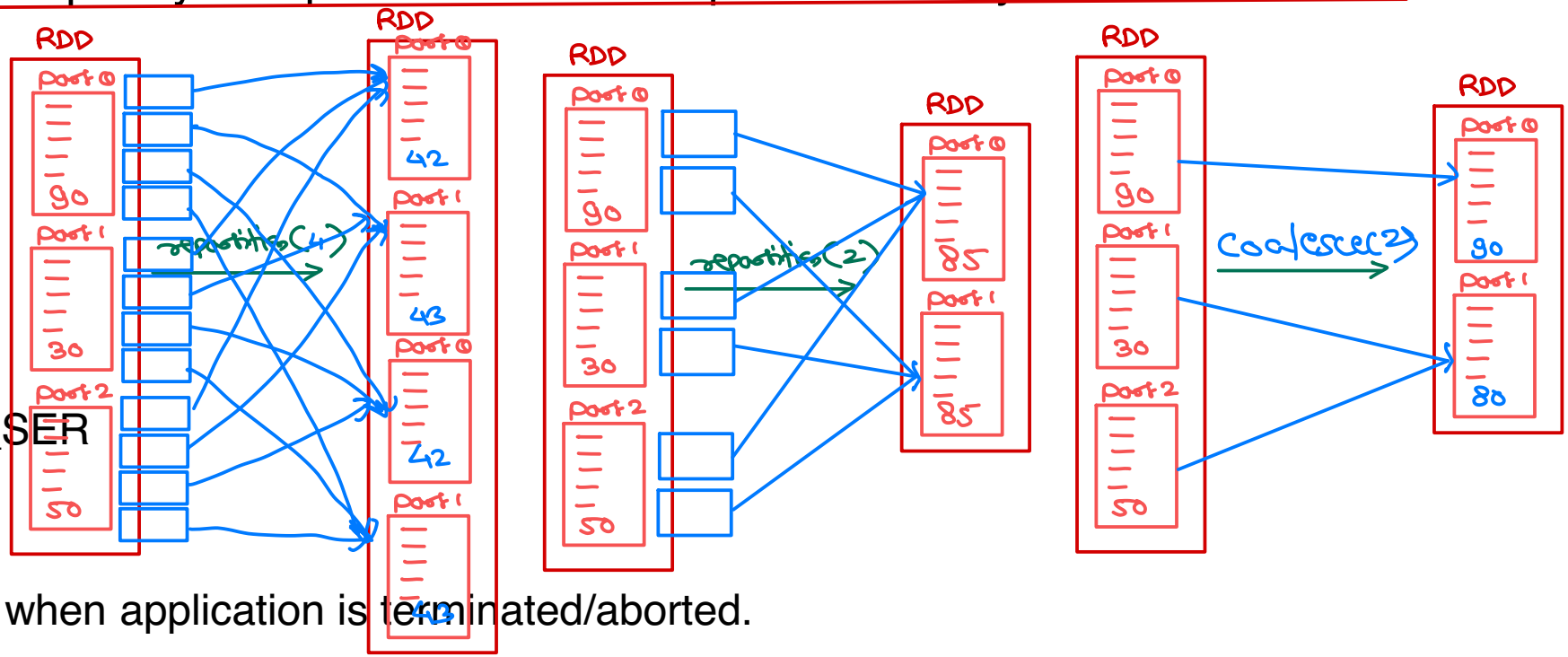
RDD Broadcast variable and Accumulators

- Broadcast variable is used to send data from driver code to all the workers.
  - Data will be wrapped in broadcast variable object which will be copied on all worker nodes (in executor processes).
  - It will accessed using brVar.value.
- Accumulators are job counters.
  - Used to collect info (counter type) from the workers.
  - They are incremented (using acc.add(1)) individually on worker nodes.
  - Finally collected on driver.



RDD advanced operations

- Repartitioning
  - repartition(n) – HashPartitioner & Heavy shuffle. Balanced across partitions.
  - coalesce(n) – Merge RDDs quickly to repartition into fewer partitions. May not be balanced.
- Caching & Persistence
  - rdd.cache()
  - rdd.persist(mode)
    - MEMORY_ONLY
    - MEMORY_ONLY_SER
    - MEMORY_AND_DISK
    - MEMORY_AND_DISK_SER
    - DISK_ONLY
  - rdd.checkpoint(dirpath)
    - recover from disk, even when application is terminated/aborted.
- Lineage
  - toDebugString()



RDD advanced operations

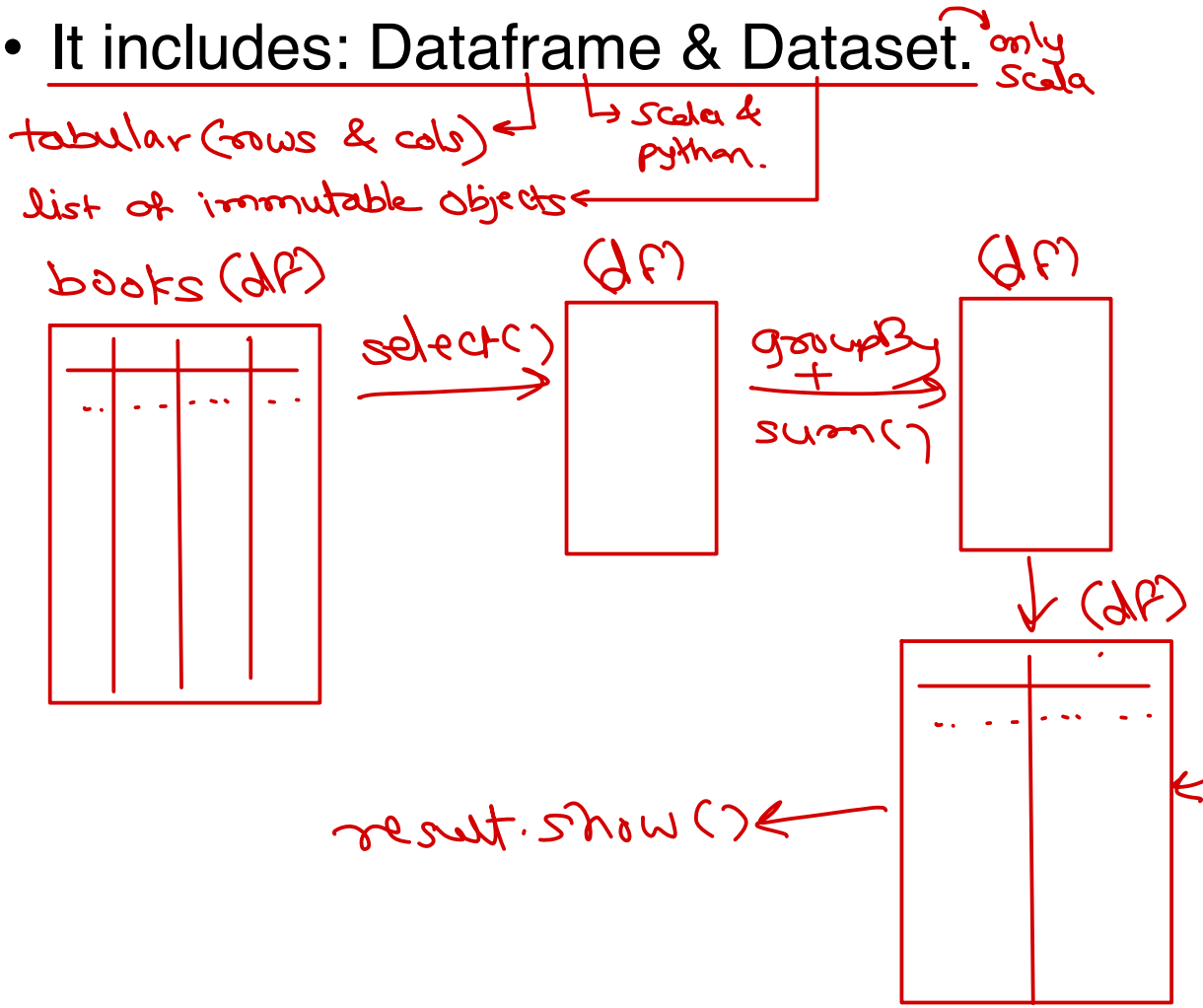
- Repartitioning
  - repartition(n) – HashPartitioner & Heavy shuffle. Balanced across partitions.
  - coalesce(n) – Merge RDDs quickly to repartition into fewer partitions. May not be balanced.
- Caching & Persistence
  - rdd.cache() → in memory
  - rdd.persist(mode)
    - MEMORY_ONLY → in memory – if rdd is too huge, few parts are discarded & recalculated when required.
    - MEMORY_ONLY_SER → like memory_only but in serialized form (compressed), needs less RAM, but more computing while deserializing
    - MEMORY_AND_DISK → in memory – if full rdd not fit in RAM then few parts are written on disk. & reloaded when required
    - MEMORY_AND_DISK_SER → like memory_and_disk but in serialized form.
    - DISK_ONLY → rdd written into disk & loaded back when required.
  - rdd.checkpoint(dirpath)
    - recover from disk, even when application is terminated/aborted.
- Lineage
  - toDebugString()

Spark – Structured API

Sunbeam Infotech

Introduction

- Spark Structured API is abstraction on Lower level concepts (RDD & DAG).
- It includes: Dataframe & Dataset.

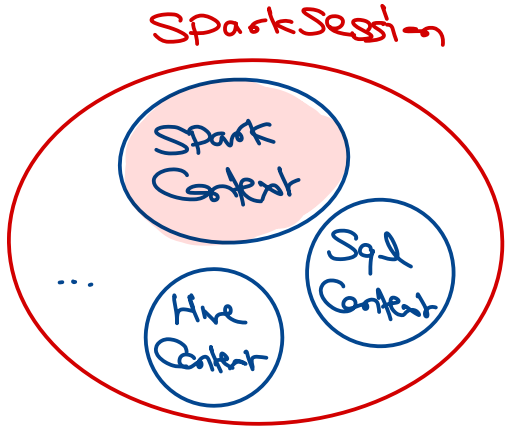



```
books = spark.read\
        .option("header", "true")\
        .option("delimiter", ",")\
        .option("inferSchema", "true")\
        .csv("/path/to/books_hdr.csv")

result = books\
        .select("subject", "price")\
        .groupBy("subject").sum("price")\
        .orderBy("subject")
```

SparkSession

- Wrapper on SparkContext. It can encapsulate additional contexts as needed e.g. SQLContext, HiveContext, StreamingContext, ...
- Spark 2.4 deprecates SparkContext.
- SparkSession is singleton i.e. one application will have single SparkSession.
- It is created using builder design pattern.

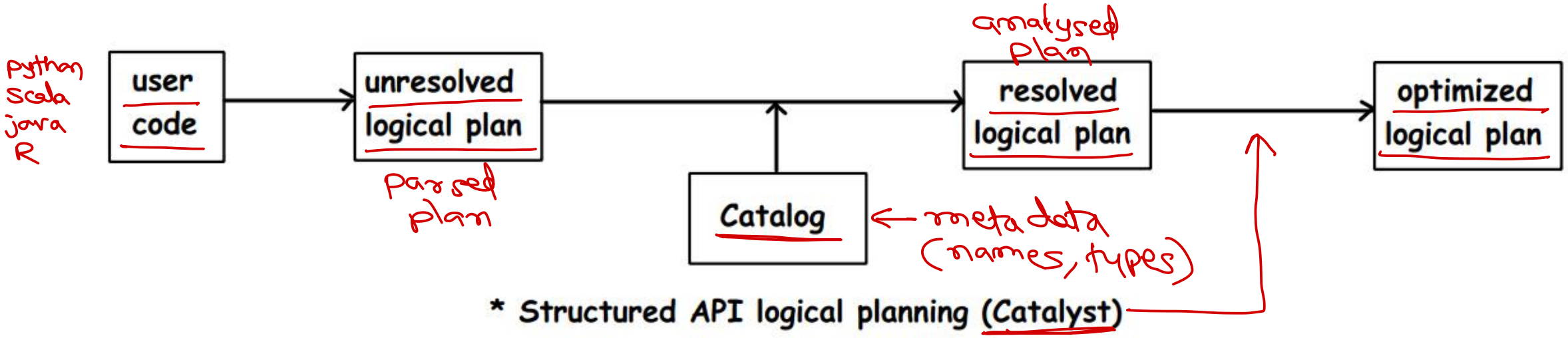


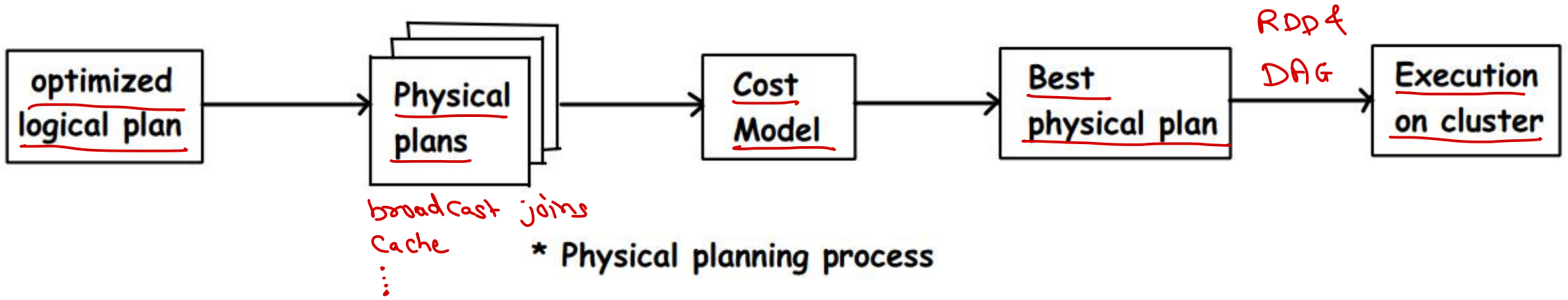

```
spark = SparkSession.builder()\
        .appName("myApp")\
        .master("local[*]")\
        .getOrCreate()
…
spark.stop()
```

Data Frames

- Created using SparkSession.
- Abstraction/wrapper on RDD.
- Similar to Pandas dataframes or R dataframes or RDBMS table. in memory
- Dataframe have structure (metadata) and rows & columns (data).
- The operations on dataframes is similar to SQL operations e.g. select(), groupBy(), orderBy(), limit(), where(), join(), ...

Spark Dataframe Execution



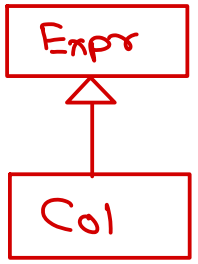


Dataframe creation

- Using DataFrameReader
  - df = spark.read.format("csv").option("key", "value").load()
  - df = spark.read.option("key", "value").csv("path")
- From Java List<Row>, RDD<Row>, NamedTuple or Dict
  - df = spark.createDataframe(data, schema)
- Schema can be inferred or can given manually.
  - df = spark.read.schema(my_schema).option("key", "value").csv("path")

Dataframe columns

- Columns are expressions.
- Expression can be column name or some arithmetic expression or some sql fn processing expression.
  - e.g. "job", "sal", "sal + comm", "sal + ifnull(comm,0)", "1 as one", "*", ...
- Selecting columns/expression
  - df.select(c1, c2, ...)
  - df.selectExpr(e1, e2, ...)
  - df.withColumn("colname", "expression") -- add extra column
- Drop column
  - df.drop("colname")



Dataframe rows

- Internally Dataframe is RDD of Row type.
- Row is StructType.
  - e.g. Row(job='CLERK', sum(sal)=4150.0, sum(comm)=None, sum(income)=4150.0)
- Individual column in Row can be accessed using index [n].

Dataframe operations

- DF operations are transformations or actions.
  - Transformations produce new dataframe.
  - Actions cause execution plan preparation & execution.

} lazy evaluation like RDD.

- Transformations
  - select(), selectExpr(), where()
  - orderBy(), sort() -- asc/desc and one/more columns
  - limit(), distinct()
  - groupBy("col").someAggOp("col")
  - join()
  - repartition(), coalesce()
- Actions
  - df.show()
  - df.first(), df.take(), df.collect()
  - df.write.format("csv").option("path", "dirpath").save(), df.write.csv("dirpath")
  - df.write.saveAsTable() → parquet format.

Spark SQL Functions

- Numeric functions
  - abs(), floor(), ceil(), round(), pow(), ...
- String functions
  - substring(), lower(), upper(), concat(), ...
- Null value functions
  - ifnull(), isnull(), ...
- Date Time functions
  - from_unixtime(), to_timestamp(), to_date(), ...
  - current_date(), current_timestamp(), ...
  - date_diff(), ...
- Aggregate functions
  - sum(), avg(), count(), min(), max(), stddev_pop(), corr(), ...
- Complex types
  - explode(), array_contains(), ...
- Window functions
  - rank(), dense_rank(), ...

Spark data formats

- Hive use SerDe to write/read data from hive table.
- Dataframes are created using DataframeReader (spark.read) and can be saved using DataframeWriter (df.write).
- Supported formats
  - csv, json, text
  - orc
    - columnar file format
    - designed & optimized for hive
  - parquet
    - columnar file format
    - designed & optimized for spark
    - default format (i.e. if no format is mentioned)
    - efficient than CSV/JSON data.
    - parquet-cli is python package to read parquet file format.
  - jdbc
    - read/write data from/to RDBMS.

Spark SQL

Sunbeam Infotech

Introduction

- Based on Spark structured API i.e. dataframes.
- Enable writing SQL queries on Spark dataframes as views/tables.
- Before Spark 2.x, SQLContext provides SQL functionality.
- Spark 2.x SparkSession encapsulate SparkContext. +SqlContext
- SparkContext use Hive metastore to maintain metadata.

abstraction = Catalog

Spark Views

- View is abstraction on spark dataframes.
- Created using df.createOrReplaceTempView("viewName")
- createOrReplaceTempView()
  - Creates view if not available.
  - If available, replace with new view.
- View treats dataframe as in memory table & create a view (like SQL view) to fire SQL queries on it.
- The temporary view is in memory only, its info not stored in metastore. It is attached to current sparkSession.

- df.createOrReplaceGlobalTempView("viewName") creates global view, which can be shared across multiple sessions.

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>