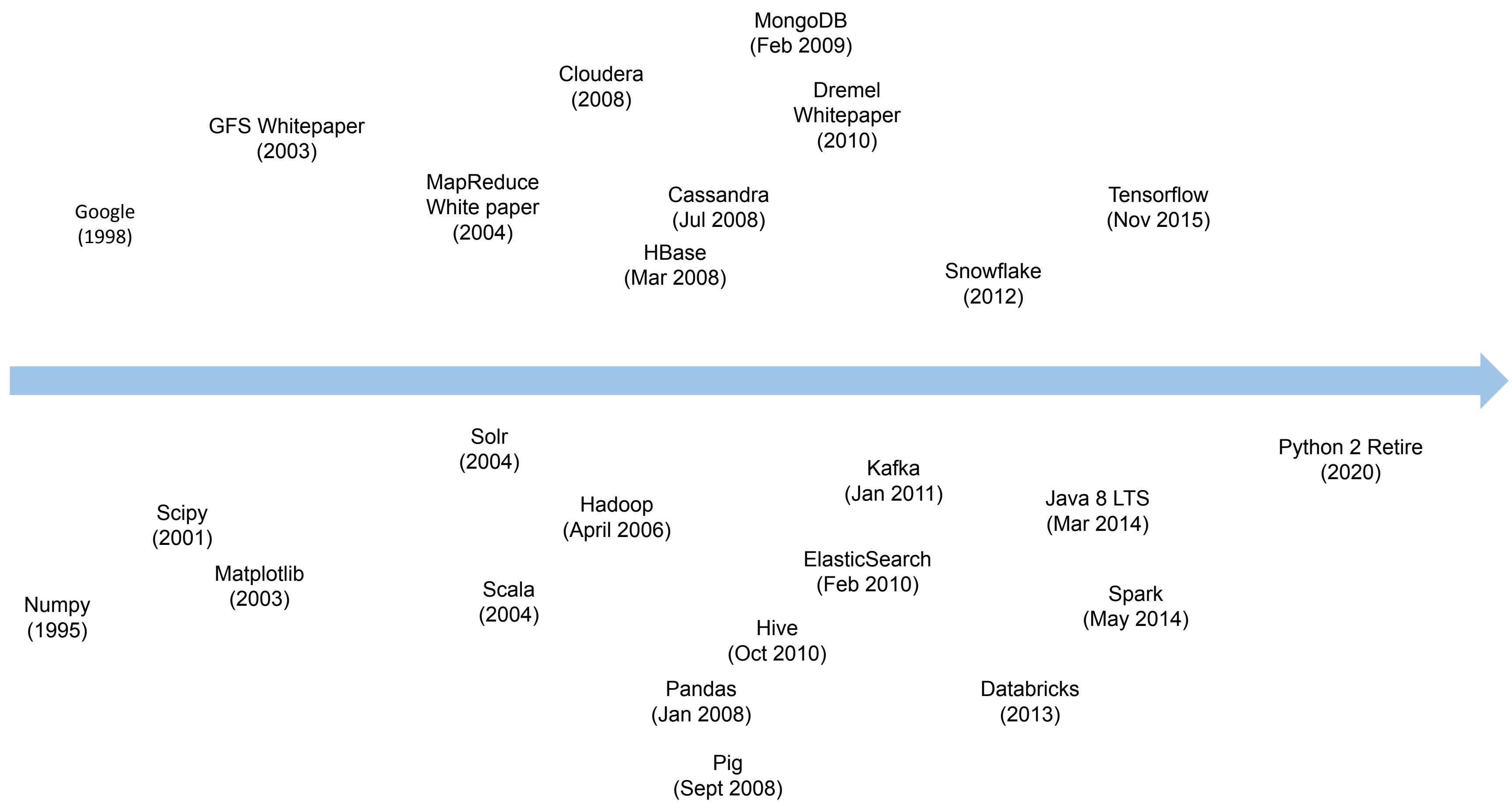
Introduction Apache Spark

	OLTP	OLAP
Access Patterns	The access pattern of an OLTP system is characterized by a high volume of small, frequent transactions that require fast response times and concurrent access by multiple users.	The access pattern of an OLAP system is characterized by fewer, larger, and more complex queries that require longer response times but provide greater analytical capabilities.
Data Model	OLTP systems typically use a normalized data model, where data is organized into multiple tables and relationships. Normalization reduces redundancy and ensures data consistency.	OLAP data models tend to be more denormalized. This should reduce the number of joins required and generally make it easier for an analyst to understand how to write their query.
Size	OLTPs tend to be smaller in terms of memory since they might only hold the current data and not historical changes.	OLAPs will be larger as they will store historical data as well as data from multiple systems.
Performance Needs	OLTPs need to have fast response times. Otherwise, end-users would be concerned that their tweet didn't go through	OLAP systems can get away with being a little slower. But if your dashboard is taking 10 minutes, DM me.

Timeline

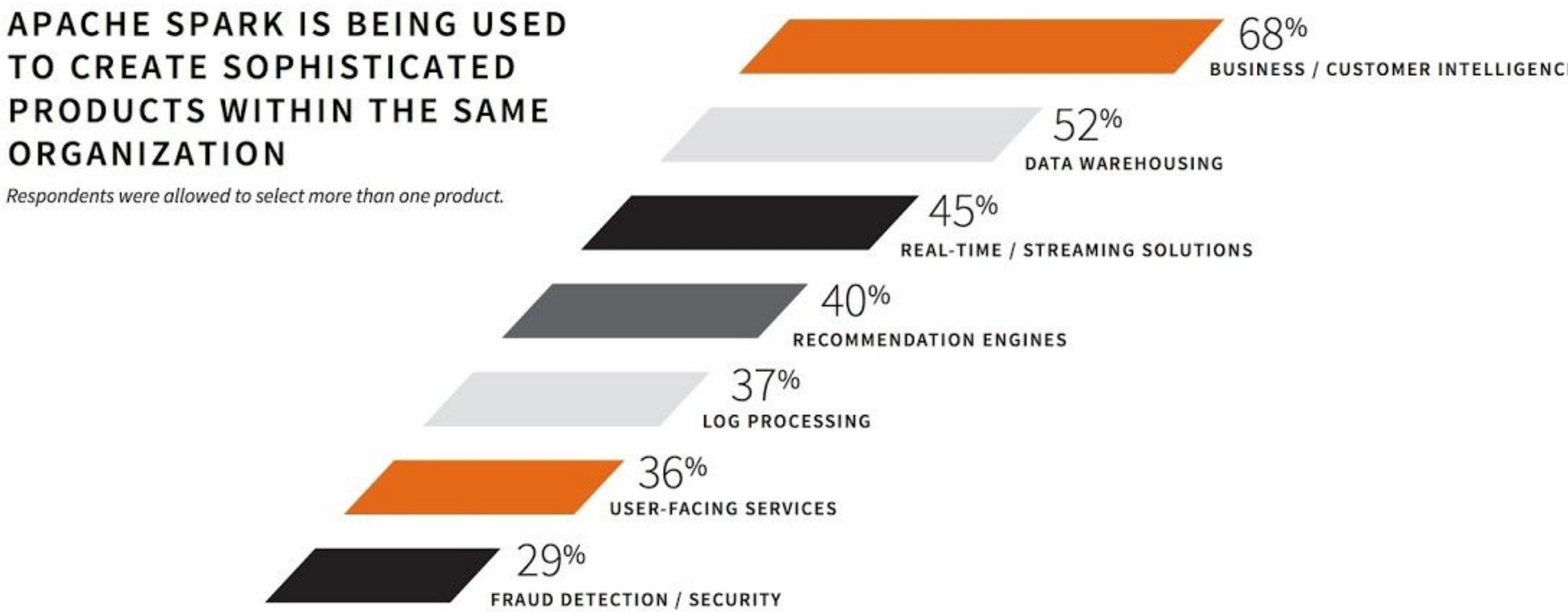

Google (1998)
GFS Whitepaper (2003)
MapReduce White paper (2004)
Cloudera (2008)
HBase (Mar 2008)
Cassandra (Jul 2008)
MongoDB (Feb 2009)
Dremel Whitepaper (2010)
Snowflake (2012)
Tensorflow (Nov 2015)
Numpy (1995)
Scipy (2001)
Matplotlib (2003)
Solr (2004)
Scala (2004)
Hadoop (April 2006)
Pandas (Jan 2008)
Pig (Sept 2008)
Hive (Oct 2010)
ElasticSearch (Feb 2010)
Kafka (Jan 2011)
Databricks (2013)
Java 8 LTS (Mar 2014)
Spark (May 2014)
Python 2 Retire (2020)

Apache Spark

- Started in 2009 as research project to create an alternative to Hadoop Mapreduce
- First release 2014 under ASF, current version 3.4.0
- Suitable for analytical workload (OLAP)
    a. Like what teradata, vertica does, but spark offers much more and for more data
- Distributed general purpose framework nicely integrated with Hadoop ecosystem
    a. It is not a replacement of Hadoop … it replaces Hadoop MR, not YARN, or HDFS
- Popular for 5 mains reasons (explained next)
    a. Speed - It is fast
    b. Various workloads with the same framework
    c. Productivity
    d. Data sources
    e. Deployment options
- Spark API doc - https://spark.apache.org/docs/latest/

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf

Common use-cases according to 2016 Survey



[2016 user survey]

A. Processing Speed - It's Fast

- In memory computation - we can cache commonly used dataset
  - not every process is in memory … you have to specify what dataset you want to place i memory
  - by default no dataset is stored in memory
  -
- Optimizers
  - DAG optimizers … it allows to write a computation graph, and spark lazily executes it based possible way … similarly to what databases do to execute a SQL
  - Catalyst optimizer - parse and execute dataframe DSL and Spark SQL
  - High performance data structure
    - RDD - Resilient distributed dataset (python, scala, java)
    - Dataframe (python, scala, R)
    - Dataset (Scala, java) *
    - DStream - (python, scala, java) supports micro batches for stream processing ; sequence of RDD
  - First three are for static data, last one is for streaming data
  - Dataframe and datasets operate on an engine called tungsten to make processing memory and process efficient

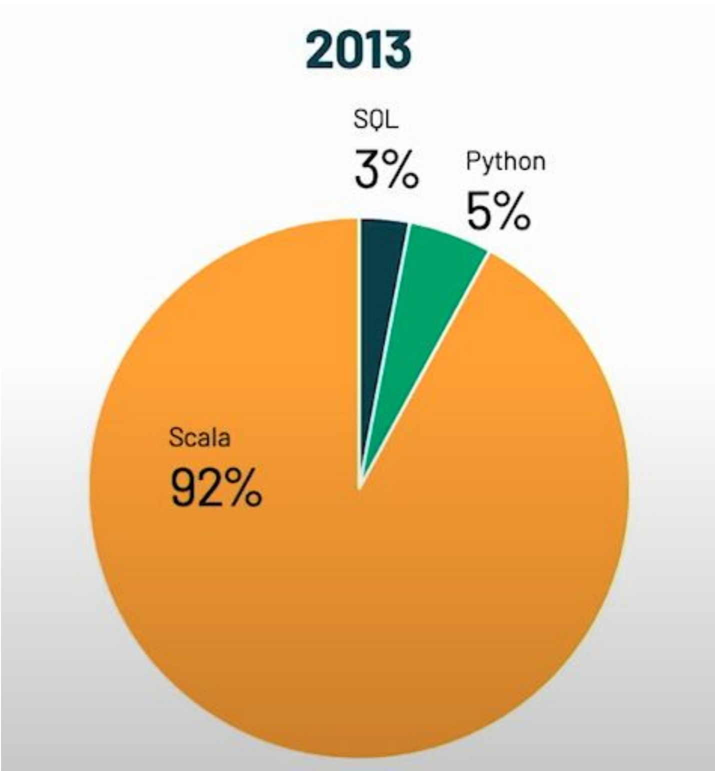
B. High productivity for the team

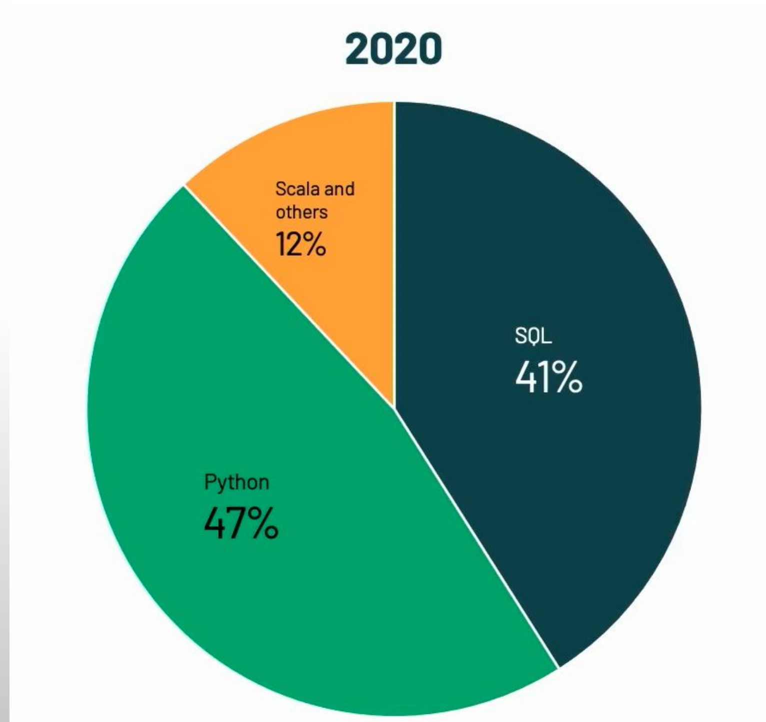
- 100+ built-in functions
- Support SQL queries (similar to Hive, but ANSI sql)
- Polyglot framework - scala *, java, python, R, SQL
    - Spark is written in scala
    - Low learning curve
    - Choose the language that best suit the requirement, for example, for visualization use Python and R, want to level Stanford NLP libraries use Java or Scala
- Interactive development - python, scala, R
    - faster iteration, rapid prototyping, data exploration
    -

Considerations behind selection of programming language

Factors
What type of use case - ETL, interactive analysis, streaming, ML, Graphx?
Is your application time aiming for strict SLA?
What is phase of the project - analysis, prototypes, deployment ready?
Are you planning to use language specific libraries like Tensorflow, Stanford NLP etc.
Do you have any existing application? Which programming language/framework that is built on?
Which programming language is more suitable from people perspective (training in a new programming language may hurt adoption)
Importance of code maintainability? (For example: Java is robust in terms of code maintainability and teamwork)

Language usage in databricks cloud

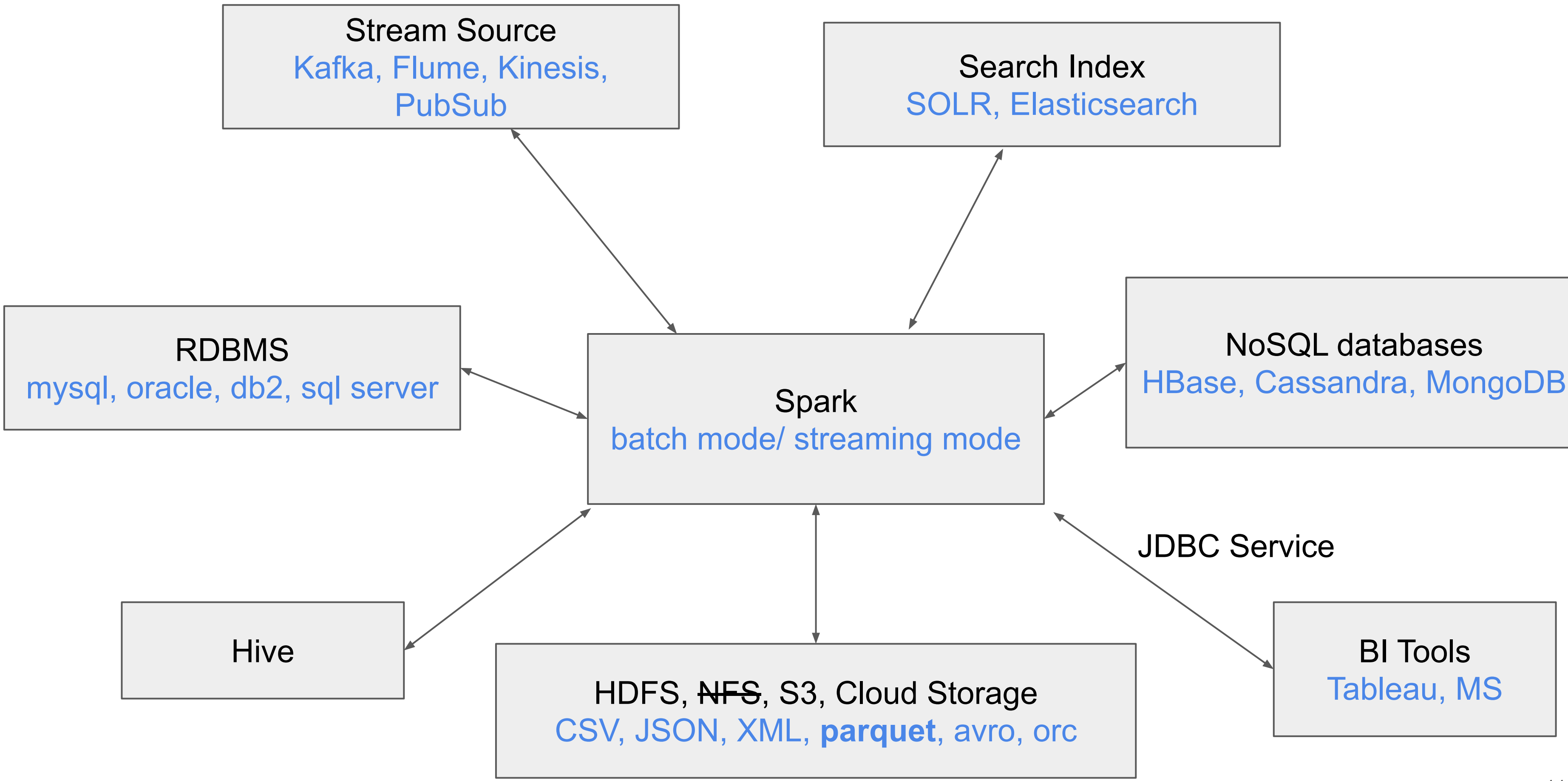




C. Wide range of workloads (OLAP)

- ETL (Extraction, Transformation, Load)/ELT
  - SQL based - similar to Hive
  - Programming language - java, scala, python etc.
- Streaming
  - Source file directory or Kafka, flume
- Interactive Analytics
  - SQL, JDBC
  - Programming - python, R, Scala, Notebook environments
- Machine Learning - Spark ML
- OLAP Graph data processing using GraphX (GraphFrame - external)

D. Data Sources



E. Deployment Option

- Hadoop YARN
- Stand alone mode - cluster manager from Spark
  - Suitable especially cloud deployments
  - Suitable for docker/kubernetes container based deployments
- Mesos
- Kubernetes
- Local Mode (single node)
  - Useful for development and prototypes

	Hadoop	Spark
Storage	HDFS	None [depends HDFS, S3 etc.]
Cluster Manager	YARN	YARN + Standalone cluster manager
Programming Framework	Map Reduce	RDD, DataFrame, Dataset, SQL
Schema Management	Hive Metastore	Hive Metastore + Delta Tables

Spark configuration

https://spark.apache.org/docs/latest/configuration.html

Setup environment for learning

- Install virtualbox https://www.virtualbox.org/wiki/Downloads
- Create Ubuntu 20.04 VM image
- Allocate 8GB ram and 4 cores to the VM

Get Started with spark (ubuntu)

Check dependencies. Required: Python [3.8, 3.9], Java 8/Java 11

```
$ python --version
$ java -version
```

If you do not have these software install

```
$ sudo apt update && sudo apt upgrade -y
$ sudo apt install python3.8
$ sudo apt-get install openjdk-8-jdk
$ sudo apt-get install openjdk-11-jdk
```

If you have multiple versions of java installed, you can set the default version using JAVA_HOME, for example

```
$ export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
```

Or you can set system default using

```
$ sudo update-alternatives --config java
```

For python, you can use virtual environment.

Configure Spark

Download spark locally from https://spark.apache.org/downloads.html. You can download the latest version of Spark.

```
wget https://downloads.apache.org/spark/spark-3.2.1/spark-3.2.1-bin-hadoop3.2.tgz
tar xf spark-3.2.1-bin-hadoop3.2.tgz
```

Move the directory to /usr/lib

```
sudo mv spark-3.2.1-bin-hadoop3.2 /usr/lib/spark

export SPARK_HOME=/usr/lib/spark
export PATH=$SPARK_HOME/bin:$PATH
export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/python/lib/py4j-0.10.9.3-src.zip:$PYTHONPATH
export PYSPARK_PYTHON=python3
export PYSPARK_DRIVER_PYTHON=python3
```

```
[Optional] To make these change permanent, you copy paste the export commands in ~/.bashrc
```

Test spark

```
$ pyspark --version
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.1
      /_/

Using Scala version 2.12.15, OpenJDK 64-Bit Server VM, 1.8.0_312
Branch HEAD
Compiled by user hgao on 2022-01-20T19:26:14Z
Revision 4f25b3f71238a00508a356591553f2dfa89f8290
Url https://github.com/apache/spark
Type --help for more information.
```

Launch Jupyter

```
export SPARK_HOME=/app/spark
export PATH=$SPARK_HOME/bin:$PATH
export PYTHONPATH=/app/spark/python:$(ls /app/spark/python/lib/py4j-*-src.zip):$PYTHONPATH
export PYSPARK_PYTHON=python3
export PYSPARK_DRIVER_PYTHON=python3
```

```
jupyter notebook --NotebookApp.open_browser=False
```

Spark session

- Every spark application has a single default global session
- You can create a multiple sessions to create multiple scopes for temporary views and UDF's

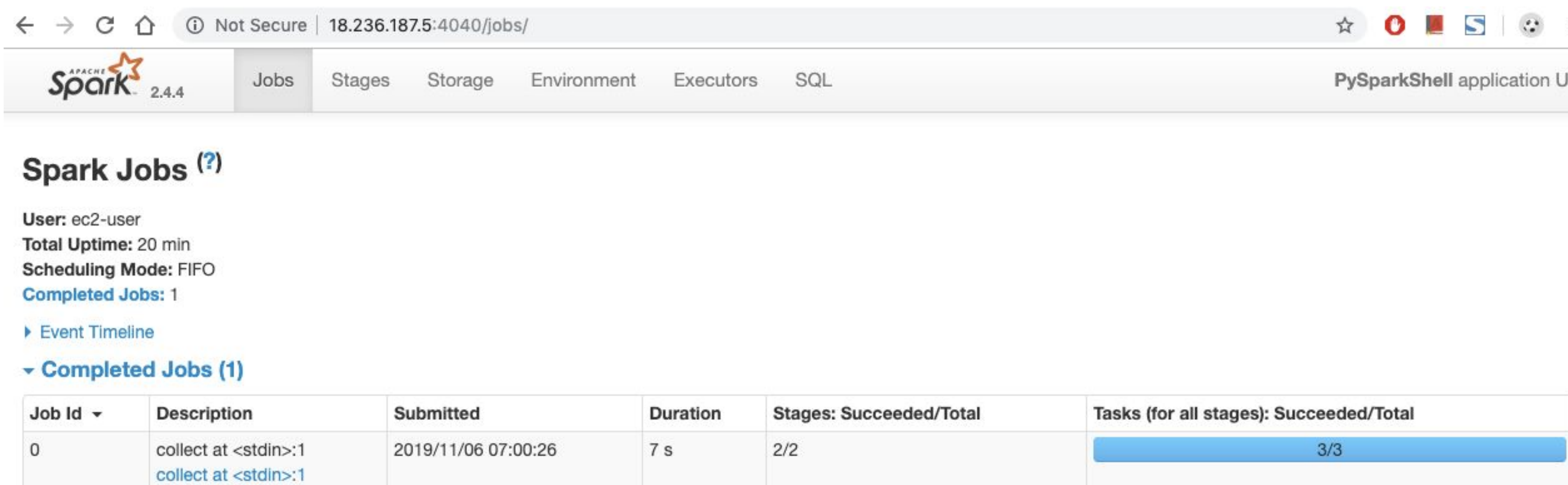
Pyspark code block to create a session

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

conf = (SparkConf()
        .setAppName("PySpark Application")
        .setIfMissing("spark.master", "local[*]")
        .setIfMissing("spark.driver.memory", "2G")
        .setIfMissing("spark.driver.cores", "2")
       )

spark = SparkSession.builder.config(conf = conf).getOrCreate()

sc = spark.sparkContext
sc.setLogLevel("WARN")


spark.sql("show tables").show()
spark.range(100).collect()
sc.parallelize(range(100)).mean()
```

Launch spark web UI

```
>>> sc.uiWebUrl
'http://ip-172-31-28-49.us-west-2.compute.internal:4040'
```

Get the server name and port where spark driver is



Not Secure | 18.236.187.5:4040/jobs/

Spark 2.4.4 | Jobs | Stages | Storage | Environment | Executors | SQL | PySparkShell application UI

Spark Jobs (?)

User: ec2-user
Total Uptime: 20 min
Scheduling Mode: FIFO
Completed Jobs: 1

Event Timeline

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <stdin>:1 collect at <stdin>:1	2019/11/06 07:00:26	7 s	2/2	3/3

https://spark.apache.org/docs/latest/web-ui.html

Open browser using the server ip and port.

Configuration https://spark.apache.org/docs/latest/configuration.html#spark-ui

Create Spark Session in scala

```scala
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession

val conf = new SparkConf()
    .setAppName(getClass.getName)
    .setIfMissing("spark.master", "local")

val spark = SparkSession
     .builder()
     .config(conf)
     .appName(getClass.getName)
     .getOrCreate()

import spark.implicits._
```

Create spark session in java

```
public static void wordCountJava8( String filename )
    {
        // Define a configuration to use to interact with Spark
        SparkConf conf = new SparkConf().setMaster("local").setAppName("Word Count App");

        // Create a Java version of the Spark Context from the configuration
        JavaSparkContext sc = new JavaSparkContext(conf);

        // Load the input data, which is a text file read from the command line
        JavaRDD<String> input = sc.textFile( filename );

        // Java 8 with lambdas: split the input string into words
        JavaRDD<String> words = input.flatMap( s -> Arrays.asList( s.split( " " ) ).iterator() );

        // Java 8 with lambdas: transform the collection of words into pairs (word and 1) and then count them
        JavaPairRDD<String, Integer> counts = words.mapToPair( t -> new Tuple2<>( t, 1 ) ).reduceByKey( (x,
y) -> x+y );

        // Save the word count back out to a text file, causing evaluation.
        counts.saveAsTextFile( "output" );
    }
```

Start zeppelin

Download Zeppelin

```
tar xf zeppelin-0.8.1-bin-all.tgz
sudo mv zeppelin-0.8.1-bin-all /usr/lib
cd /usr/lib/zeppelin-0.8.1-bin-all/
```

Add the following lines to the file ~/.bashrc

```
export JAVA_HOME=/usr/java/jdk1.8.0_201
export SPARK_HOME=/usr/lib/spark-2.4.0-bin-hadoop2.7
export ZEPPELIN_HOME=/usr/lib/zeppelin-0.8.1-bin-all
```

Start Zeppelin

```
cd ~ && sudo /usr/lib/zeppelin-0.8.1-bin-all/bin/zeppelin-daemon.sh start
```

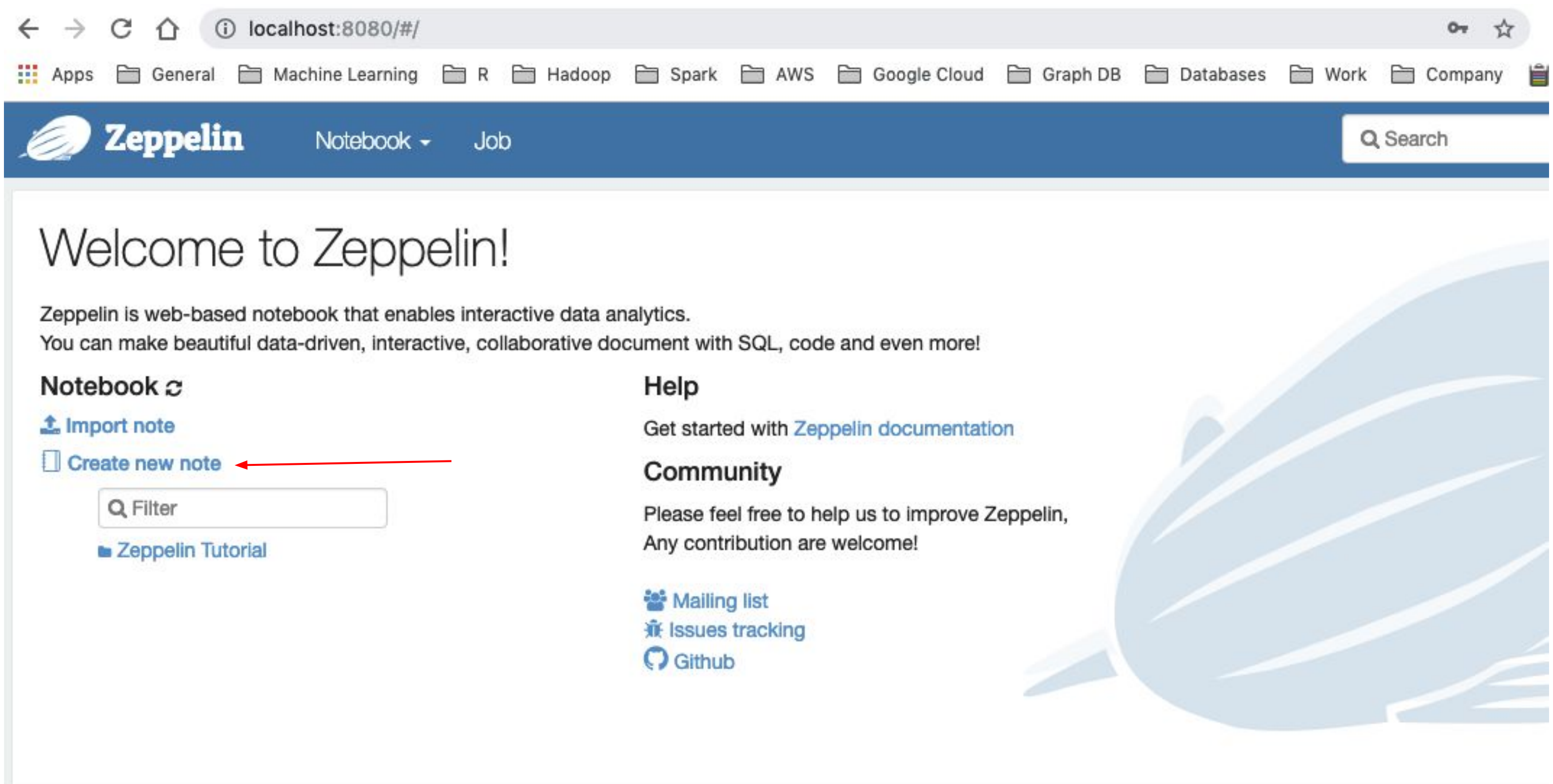
Read data from S3

```
$ pyspark \
--packages com.amazonaws:aws-java-sdk-pom:1.10.34,org.apache.hadoop:hadoop-aws:2.6.0

hadoopConf = sc._jsc.hadoopConfiguration()
hadoopConf.set("fs.s3.impl", "org.apache.hadoop.fs.s3native.NativeS3FileSystem")
hadoopConf.set("fs.s3.awsAccessKeyId", "AKIA...")
hadoopConf.set("fs.s3.awsSecretAccessKey", "vl5W...")

rdd = sc.textFile("s3://data.einext.com/stocks/stocks.csv.gz")
rdd.take(10)
```

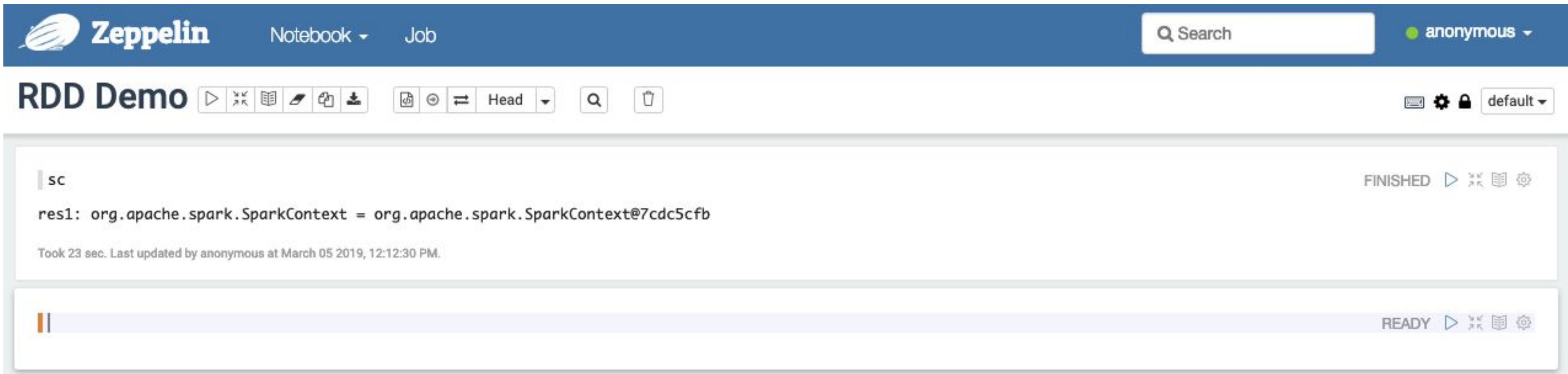
Optimal file size for S3

https://medium.com/bigspark/compaction-merge-of-small-parquet-files-bef60847e60b

Open Zeppelin notebook



Create a notebook and run the following as in screen



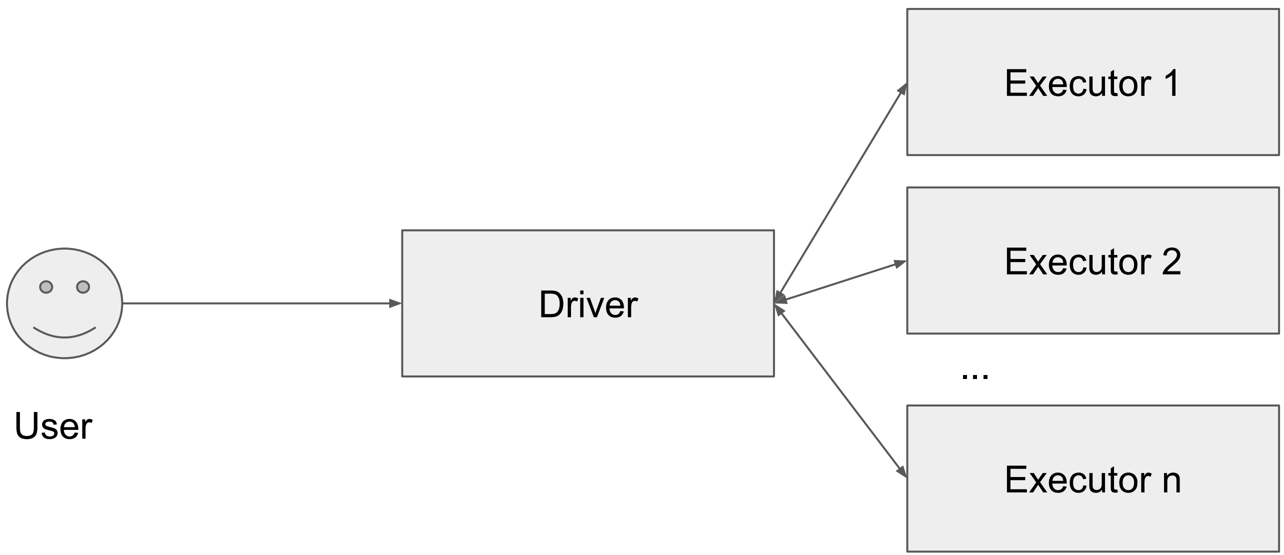
Hadoop cluster

Cluster of machines running Hadoop at Yahoo! (Source: Yahoo!)

Example: Hadoop @ LinkedIn

- 10+ hadoop clusters running on bare metal (no VM) systems
- Largest Hadoop cluster is 7000+ servers with capacity of 400+PB, half million v-CPU cores, 1.6PB memory
- Multi-clusters with 4000+ nodes
- Average R/W throughput 600+ GB/sec each
- 300K+ daily jobs with hundreds of millions containers
- 1000+ container allocations per second
- QPS: HDFS namenode RPC: 100K+, LDAP 150K+, KDC 5K+, DNS 95K+
- Inbound and outbound network traffic is 15TB/sec

Spark Architecture - Scala/Java RDD execution

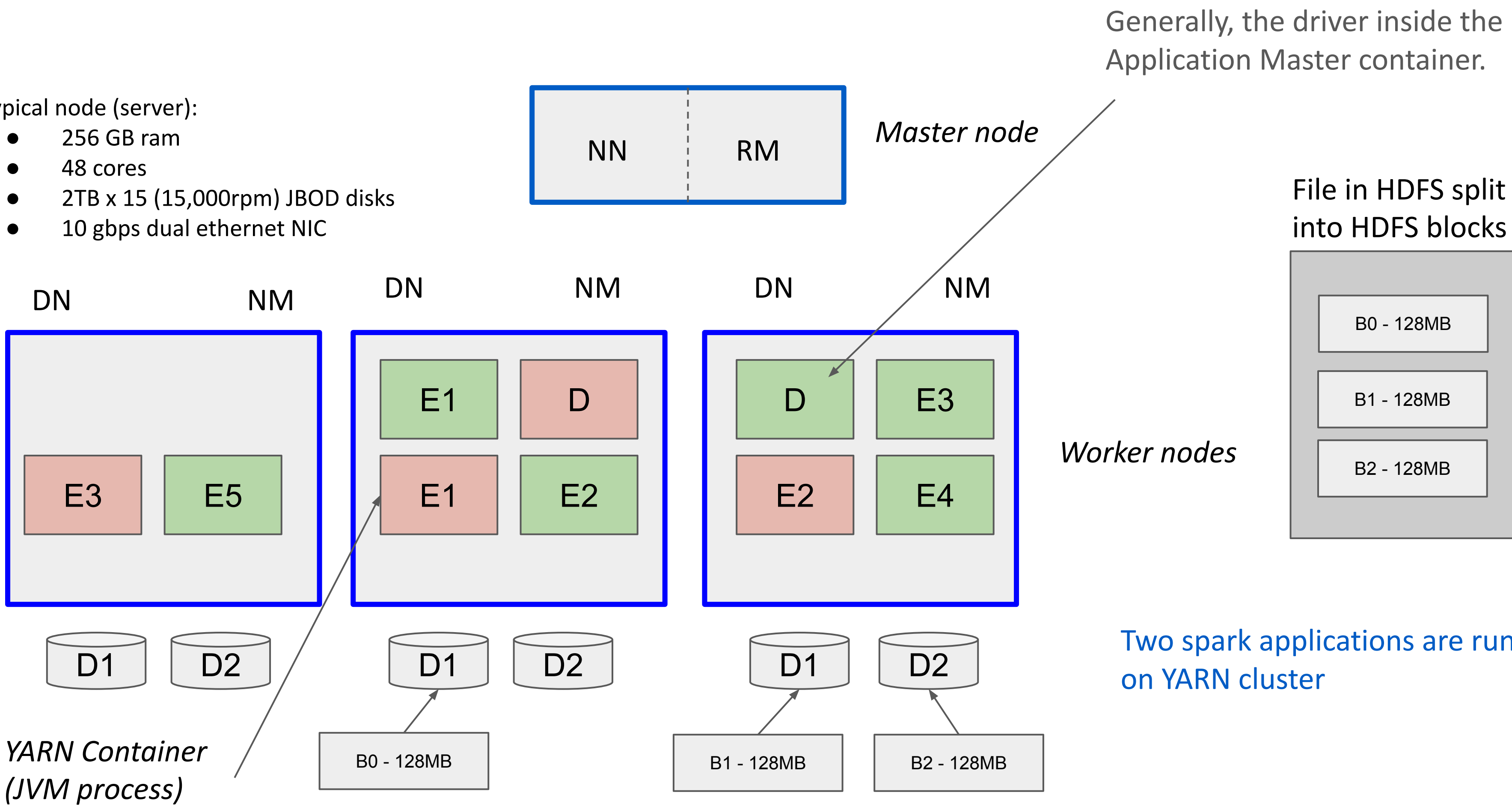


- One spark session consists of one driver and one or many executors
- Driver is the supervisor and executors are real workhorse
- Drive and executor are JVM processes
- You can configure RAM and v-CPU core per driver and executors
- Number of v-cores per executor ⇒ Number of parallel tasks in an executor
- Each of driver and executor runs on YARN container, if spark is deployed on YARN
- Number of executors can be specified during launch or can depend on load (speculative execution)

Spark sessions on Hadoop YARN

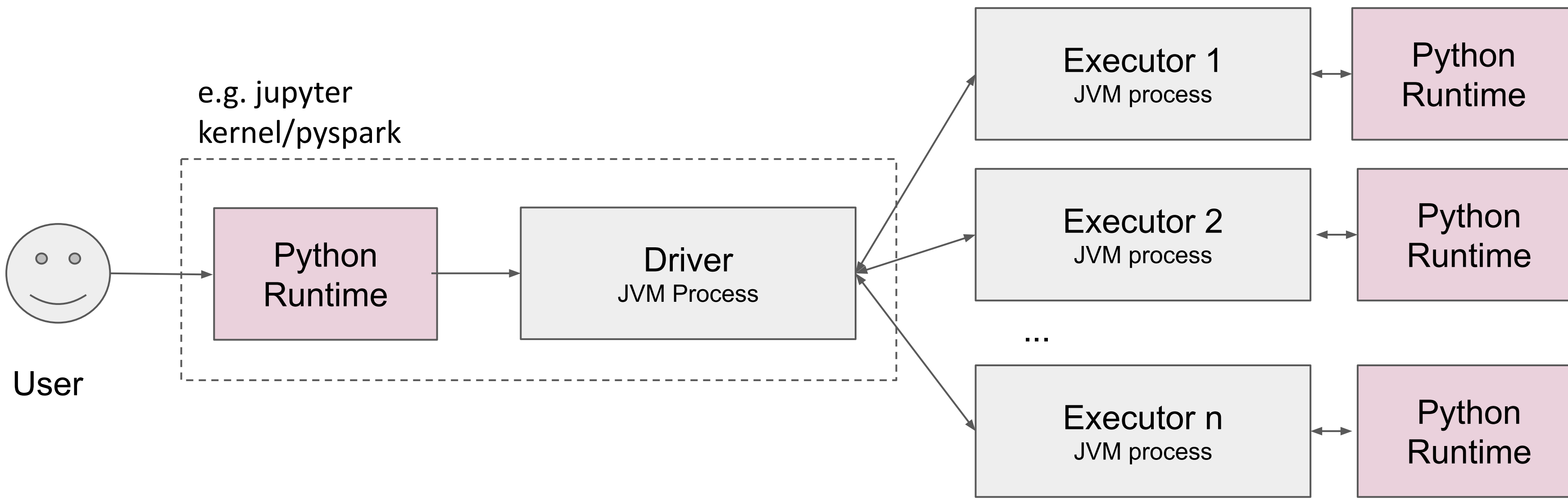
Typical node (server):

- 256 GB ram
- 48 cores
- 2TB x 15 (15,000rpm) JBOD disks
- 10 gbps dual ethernet NIC



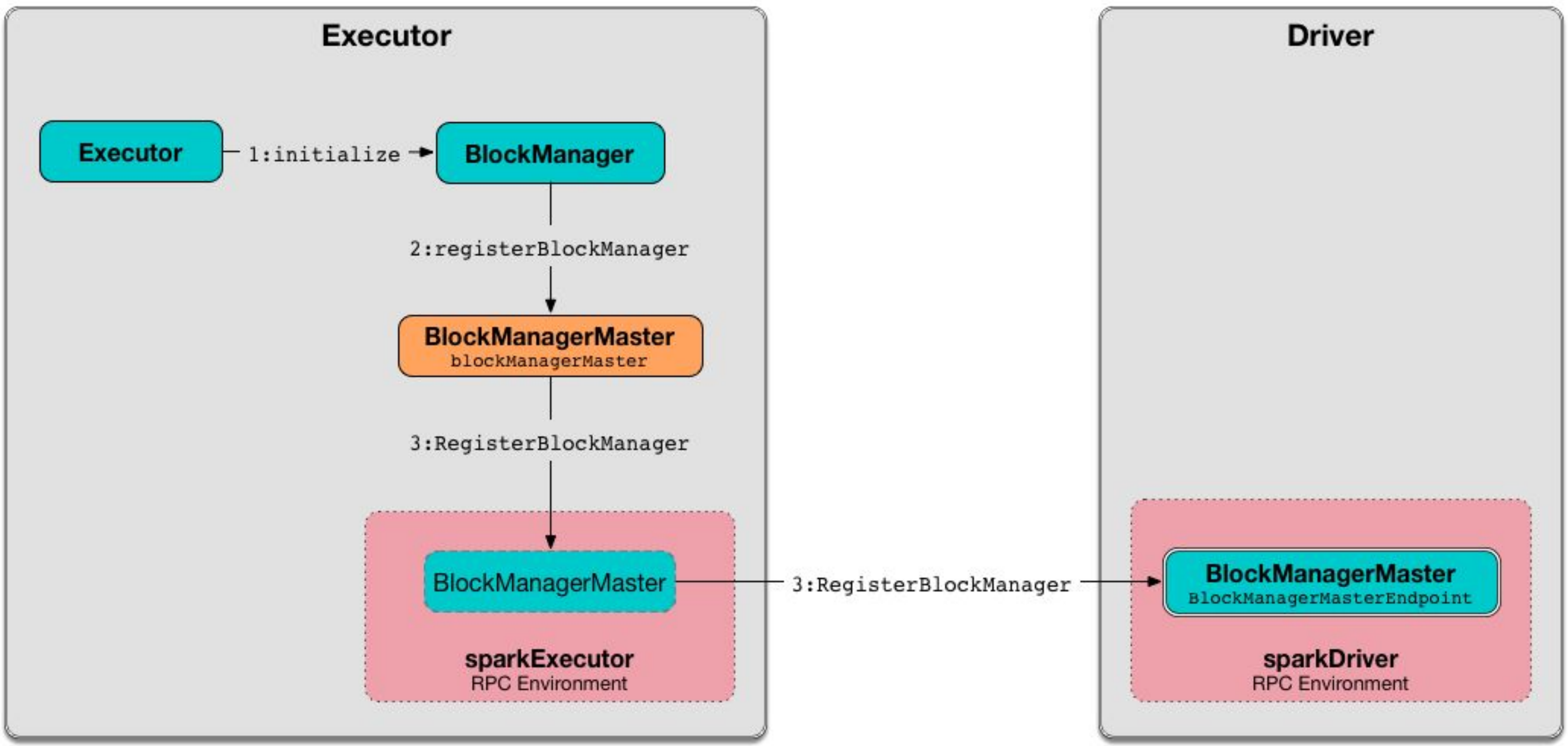
Two spark applications are running on YARN cluster

Spark Architecture - RDD ops in **Pyspark**

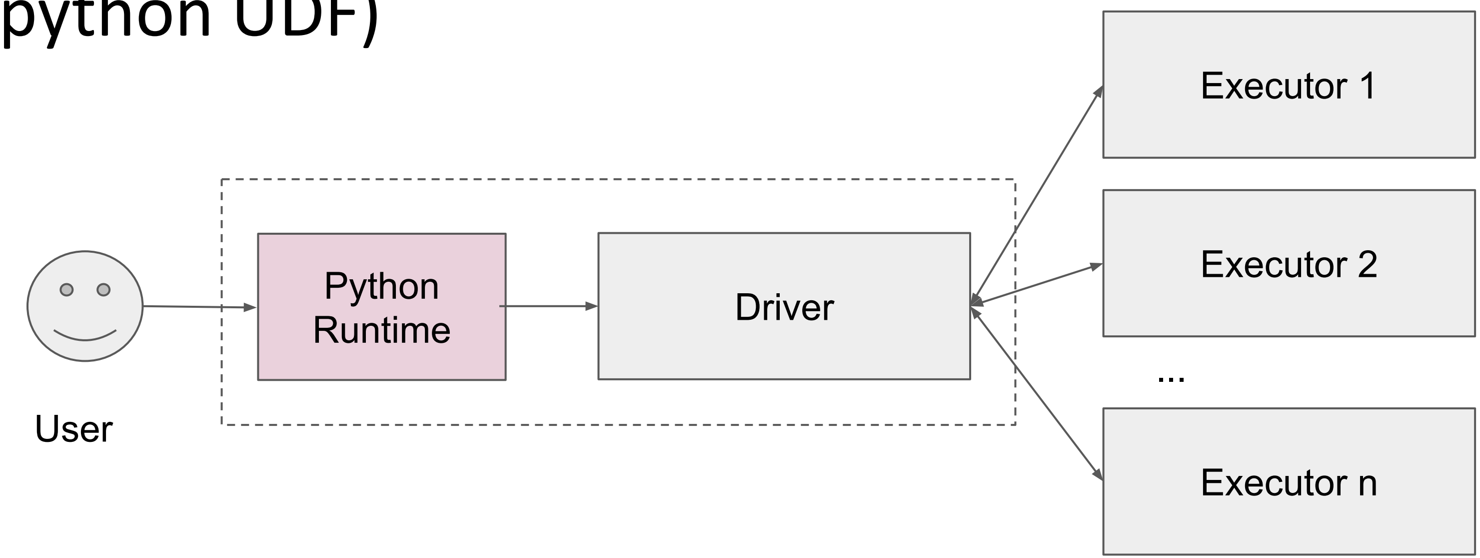


- When you use pyspark for RDD operations, JVM based drivers and executors will perform normal data transfer operations with the cluster, but caching and processing happens in Python run time.
- So at the executor level, data have to be copied back and forth between JVM and python runtime, making the operations relatively slower.
- Interface between python and jvm process is managed by py4j

Block Manager (Internal Details)

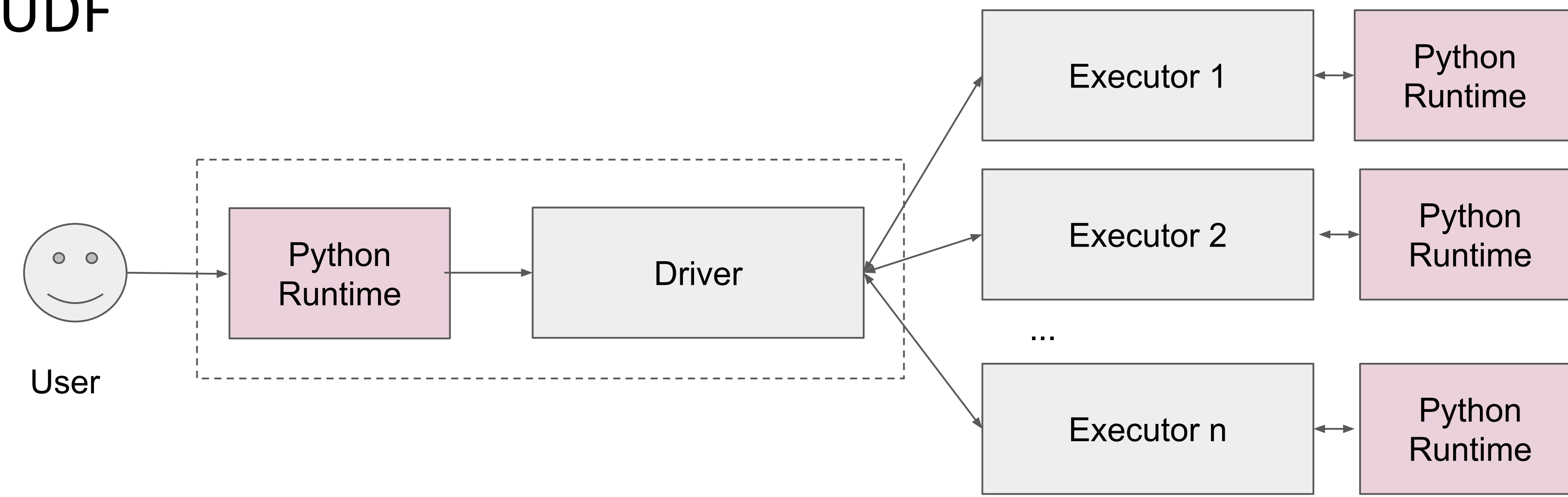


Spark Architecture - DataFrame ops using Pyspark (no python UDF)



- For the DataFrame operations in pyspark. python only invokes the built-in methods for DataFrame.
- Python runtime sends a SQL query to driver, which parses and execute the queries with the help of executors
- Dataframe operations are executed within JVM

Spark Architecture - DataFrame ops in Pyspark + python UDF



- The UDF defined in python is executed in python kernel. Situation becomes similar to RDD operations using Python.
- While in many use cases, the convenience of python will out weight the extra time the processing needs.
- In time sensitive application, write the UDF in scala or Java and call those UDF from pyspark. UDF operates within the executor JVM. Here is example.

Pyspark

	JVM	Python runtime
RDD::filter	✓	✓
DataFrame::where	✓	
DataFrame::where calls a python UDF	✓	✓

Create a standalone cluster

Start master

```
$ cd $SPARK_HOME
$ sbin/start-master.sh
```

Open browser http://server01:8080

Start worker service on each machine

```
$ cd $SPARK_HOME
$ sbin/start-slave.sh spark://server01:7077
```

Sample Job Mission

```
bin/spark-submit --master spark://einext08:7077 --class exercise.rdd.MovieLens --packages
net.sourceforge.argparse4j:argparse4j:0.9.0  SparkDemo-1.0-SNAPSHOT.jar -m
/data/ml-latest-small/movies.csv -r /data/ml-latest-small/ratings.csv --output /tmp/movielense_output
```

History server

```
# File: // $SPARK_HOME/conf/spark-defaults.conf
spark.eventLog.enabled             true
spark.eventLog.dir                 /tmp/spark-event-logs
spark.history.fs.logDirectory      /tmp/spark-event-logs
spark.serializer                   org.apache.spark.serializer.KryoSerializer
spark.driver.memory                2g
spark.executor.memory              4g
```

```
mkdir -p /tmp/spark-event-logs
```

Start history service

```
$ cd $SPARK_HOME
$ sbin/start-history-server.sh
```

http://<history server>:18080

RDD - resilient distributed dataset

- Standard RDD
- PairRDD
- DoubleRDD

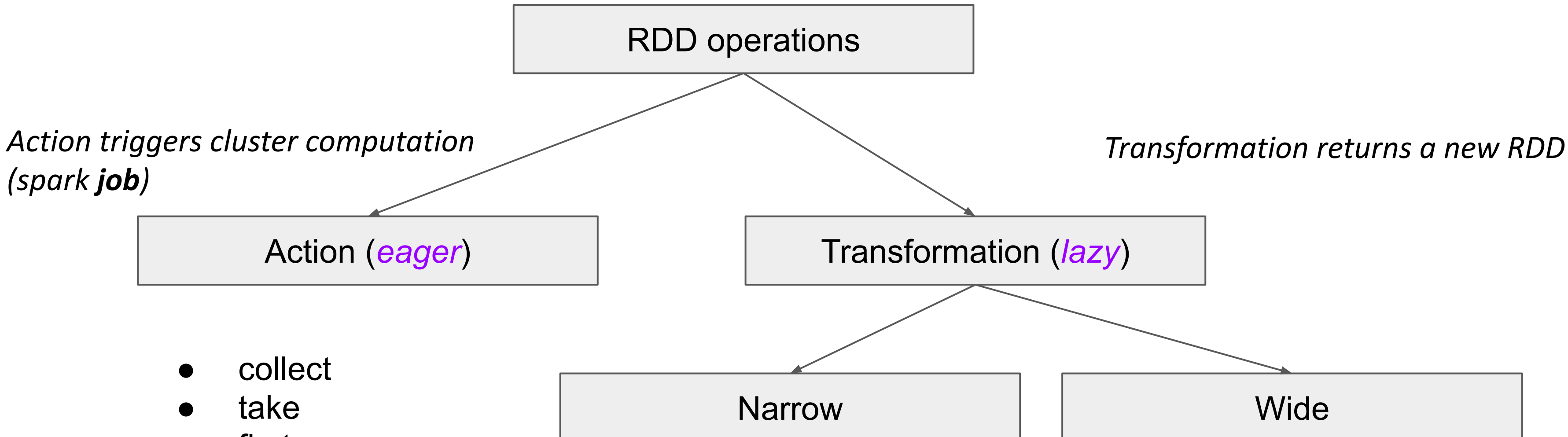
https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaRDD.html

RDD - Resilient Distributed Dataset

There are two types of operations

- **transformation** - map, filter, groupBy etc.
  - a transformation on RDD returns a new RDD
  - you look at it as view in the database
  - there are two types are transformation - wide (requires data shuffling across nodes) and narrow
  - no computation happens, it only creates a DAG
  - The resulting RDD remembers the transformation logic that helps to materialize from the parent RDD
- **action** - count, take, collect, saveAsTextFile
  - when you perform the action, cluster execution happens … spark calls it a job

Spark RDD operation types



Action:

- collect
- take
- first
- count
- saveAsTextFile
- foreach
- **foreachPartition**

Narrow:

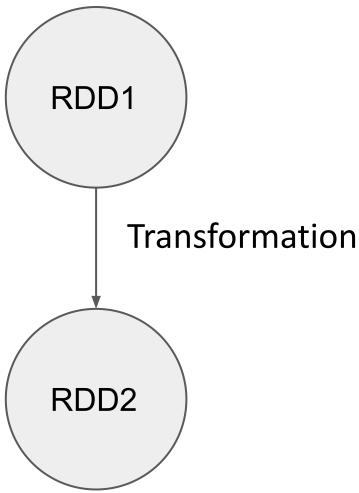
- map
- filter
- mapValues
- keyBy
- **mapPartitions**
- union

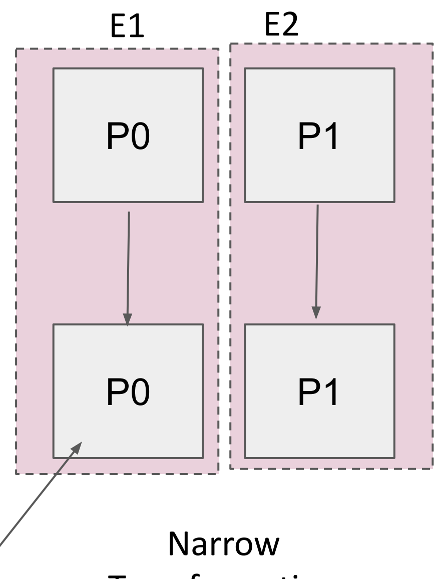
Wide:

- distinct
- sortBy
- groupByKey
- reduceByKey
- join *
- intersection

https://spark.apache.org/docs/latest/api/java/

Narrow vs wide operations

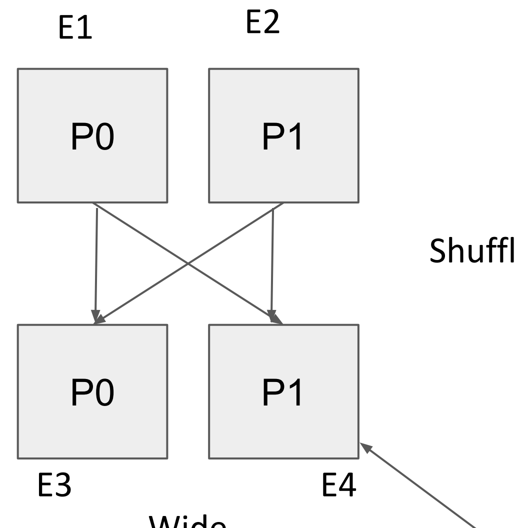




Each output record depends on values from a single partition

Narrow Transformation

- map (1:1)
- filter (1:0-1)
- flatMap(1:0-m)
- mapValues (m:1)
- keyBy (1:1)
- mapPartitions

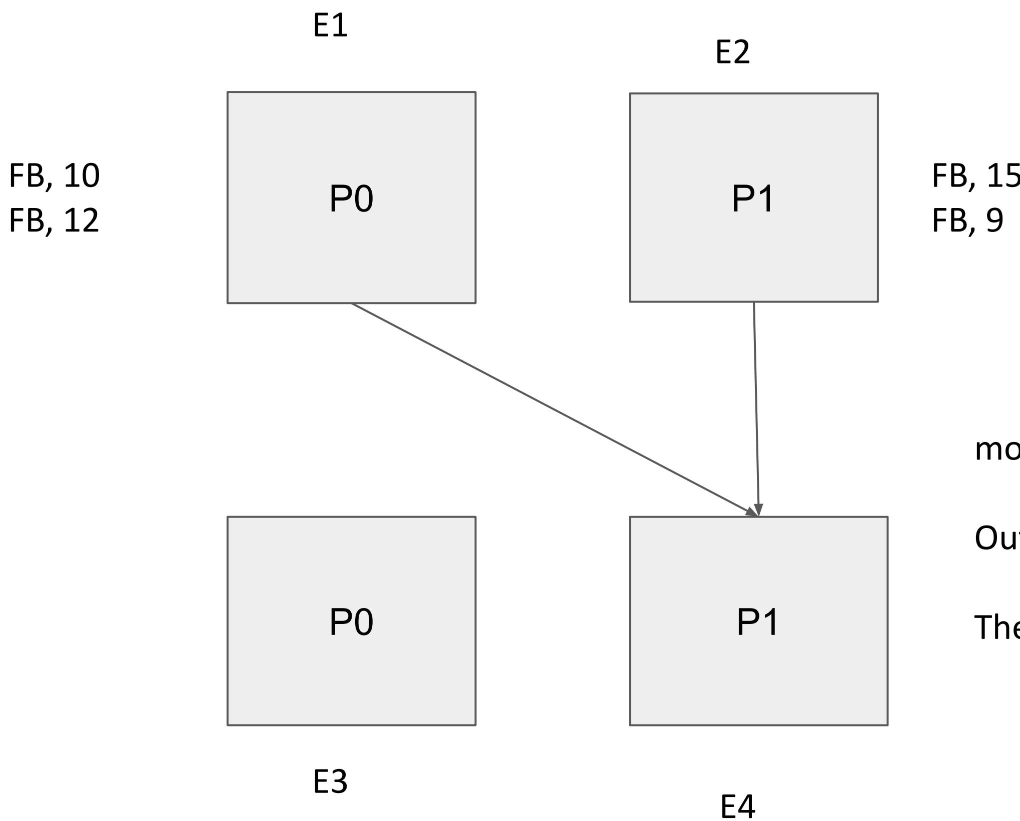


Wide Transformation

- groupByKey
- distinct
- sortBy
- reduceByKey
- join
- intersection

Each output record depends on values fro
multiple partitions

Hash Partitioner controls the target partition in case of wide transformation

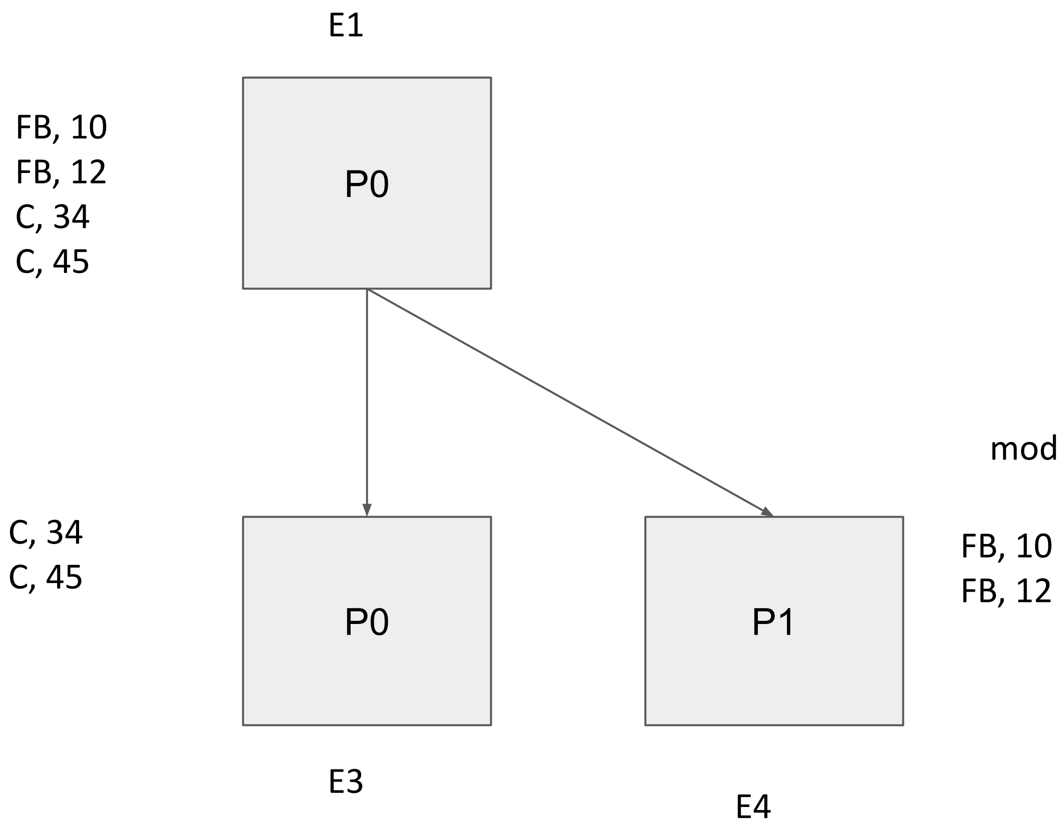


mod(hashcode(key), <num of partitions>)

Output is considered as the target partition id

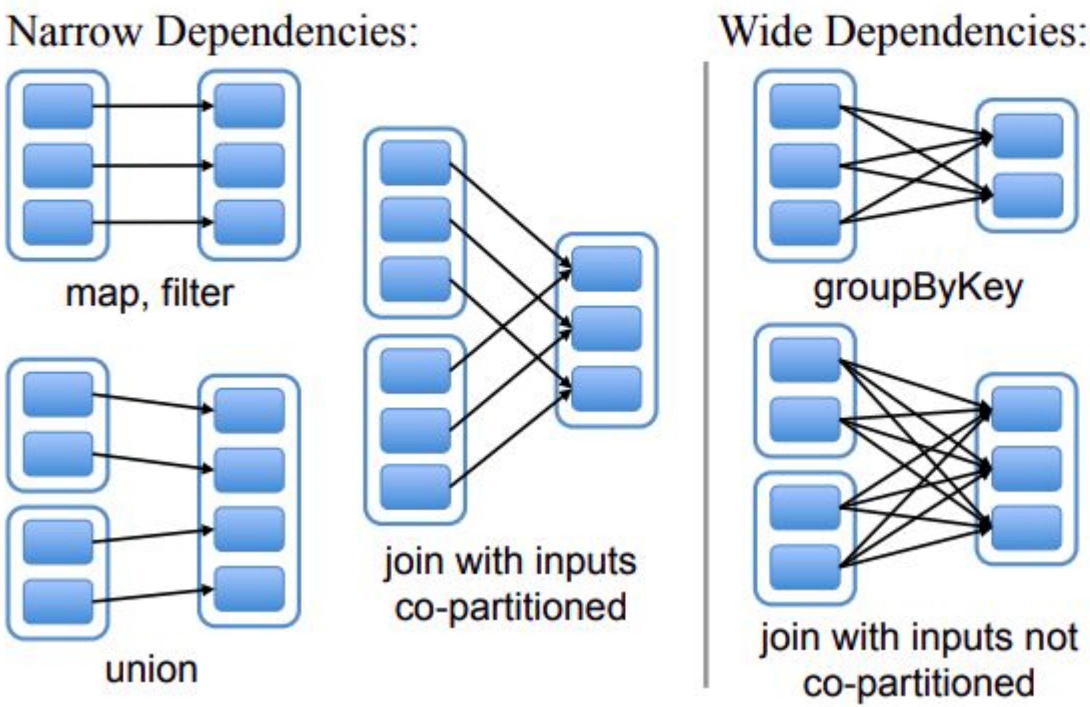
The keys must be hashable

Hash Partitioner controls the target partition

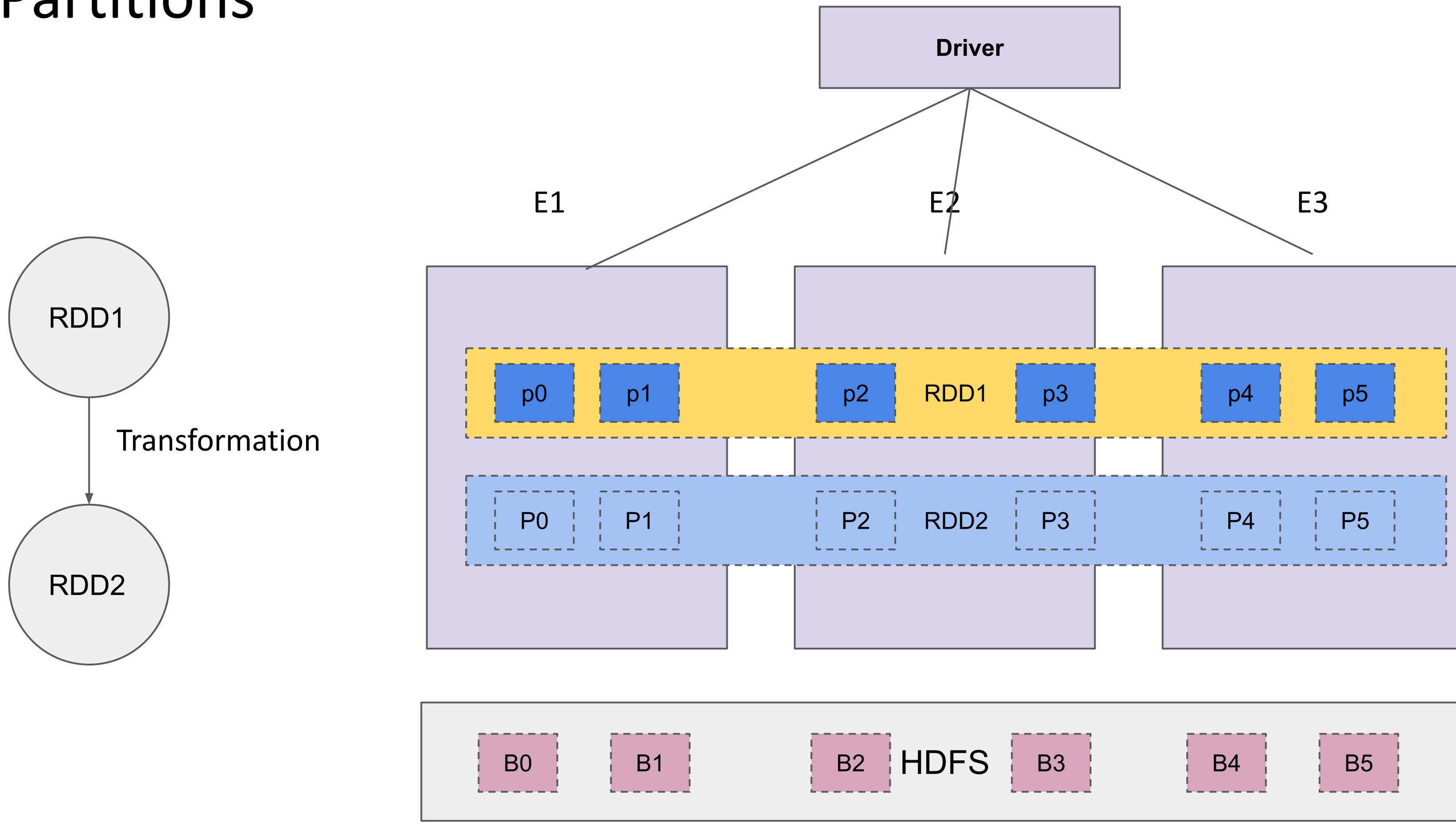


mod(hashcode(key), <num of partitions>)

Narrow vs Wide



Partitions



RDD1
Transformation
RDD2
Driver
E1
E2
E3
p0
p1
p2
RDD1
p3
p4
p5
P0
P1
P2
RDD2
P3
P4
P5
B0
B1
B2
HDFS
B3
B4
B5

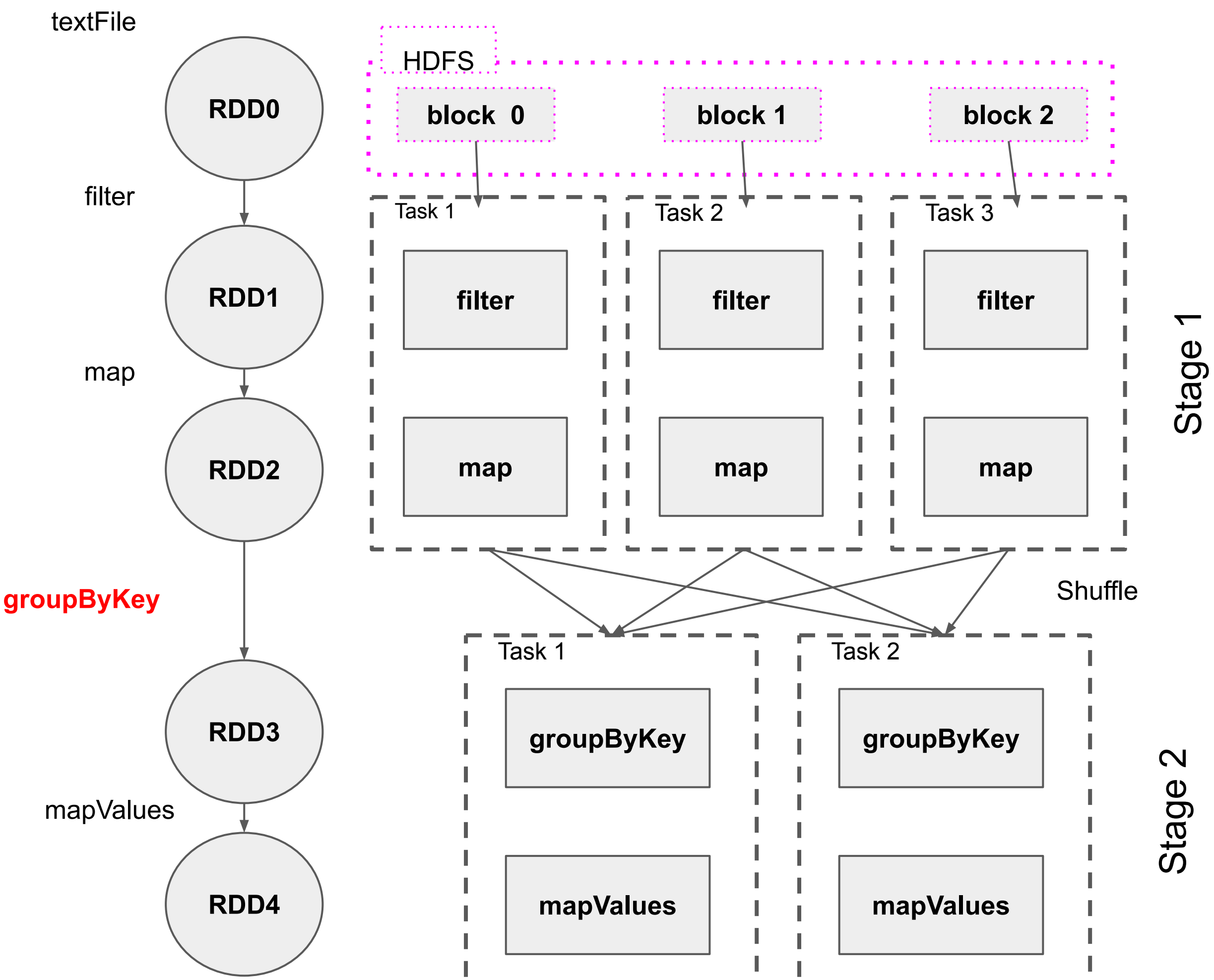
Get count of records of each partitions

```
def find_partition_size(rdd):
    def parition_size(p):
        count = 0
        for _ in p:
            count += 1
        return count
    return rdd.mapPartitions(lambda p: [parition_size(p)]).collect()
```

Find size of data on HDFS

```
def find_hdfs_size(path):
    hadoop = spark._jvm.org.apache.hadoop
    fs = hadoop.fs.FileSystem
    conf = hadoop.conf.Configuration()
    path = hadoop.fs.Path(path)
    return fs.get(conf).getContentSummary(path).getLength()

find_hdfs_size('/user/ubuntu/weblogs/')
```

Session, DAG, Data lineage, Job, Stage, Task



Task = JVM process
Allocate RAM and CPU

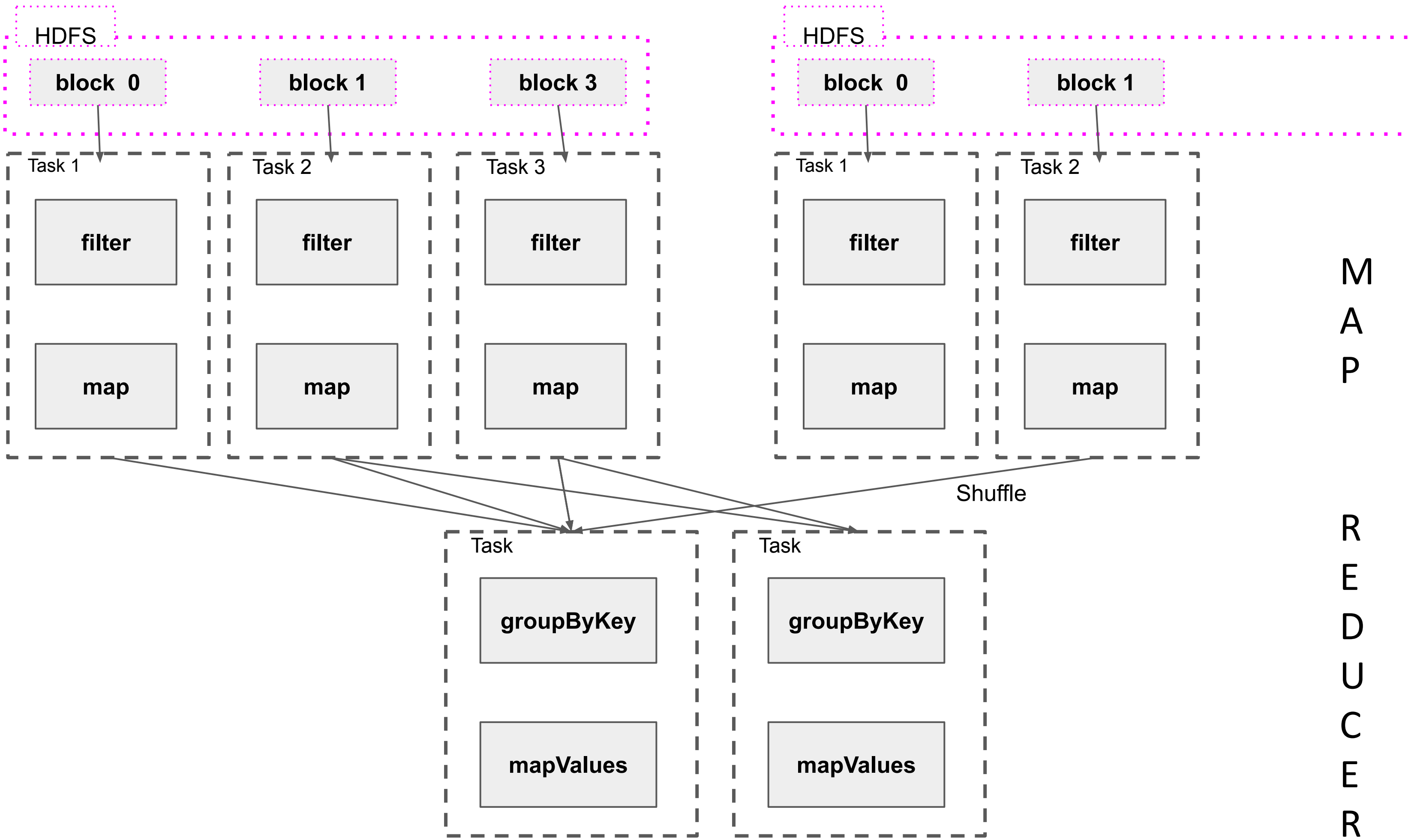
Stage is a collection of consecutive narrow transformations

- Key determines the target partition in case of wide operation
- Partitioner converts the key into partitiner Id, by taking hash(key) % num_partitions

Session, Job, Stage, Tasks

- When you create a sparkcontext, a session starts
- An action on RDD creates a job. A spark session (or application) can run multiple jobs. For example, for an rdd, you can call count, collection methods.
- By default spark executes the jobs sequentially. To run multiple jobs in parallel, you need to submit the jobs through multiple threads.
- A job consists of multiple stages
    - number of stages = num of wide operations + 1
- *Within a job, the stages are executed sequentially*
- Each stage is a group of tasks.
    - Number of tasks = the number of partitions of the RDD
- Tasks within a stage run in parallel.
- Tasks within a stage do not share data, but tasks across two consecutive stages can share data
- Executors run tasks
    - an executor can run one or more tasks simultaneously
    - An executor can run as many tasks as the numbers of "cores" (`spark.executor.core`) allocated to it at a given point of time.
    - If we allocate 4 cores per executors and use 10 executors, then total parallel tasks = 4 x 10 = 40

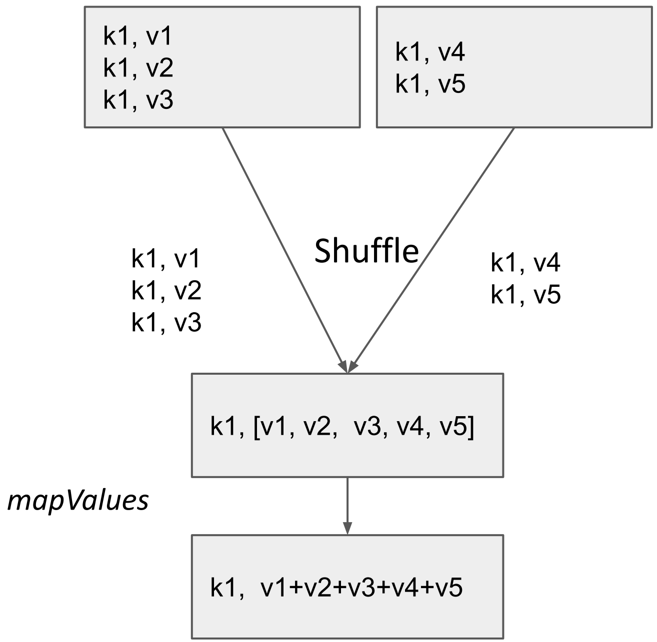
Join


HDFS
block 0
block 1
block 3
Task 1
filter
map
Task 2
filter
map
Task 3
filter
map
HDFS
block 0
block 1
Task 1
filter
map
Task 2
filter
map
Shuffle
Task
groupByKey
mapValues
Task
groupByKey
mapValues
M
A
P
R
E
D
U
C
E
R

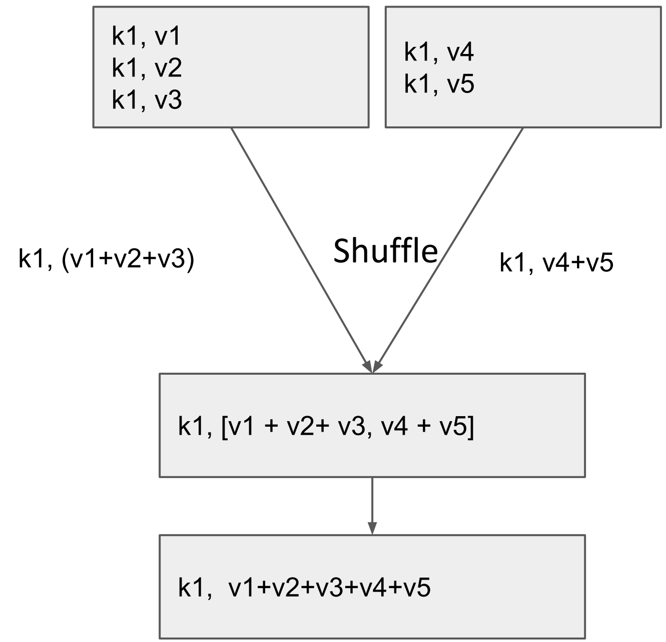
Group by Key vs Reduce By Key

Use case: compute sum for each key

More efficient because less shuffle



Group By Key



Reduce By Key

$$3 + 4 + 5 + 6$$

$$((3 + 4) + 5) + 6$$

Conditions for reduceByKey

Commutative

$f(a, b) = f(b, a)$

Associative

$f(a, f(b, c)) = f(f(a, b), c)$

CombineByKey

`RDD.combineByKey`(*createCombiner*, *mergeValue*, *mergeCombiners*, *numPartitions=None*, *partitionFunc=<function portable_hash>*)

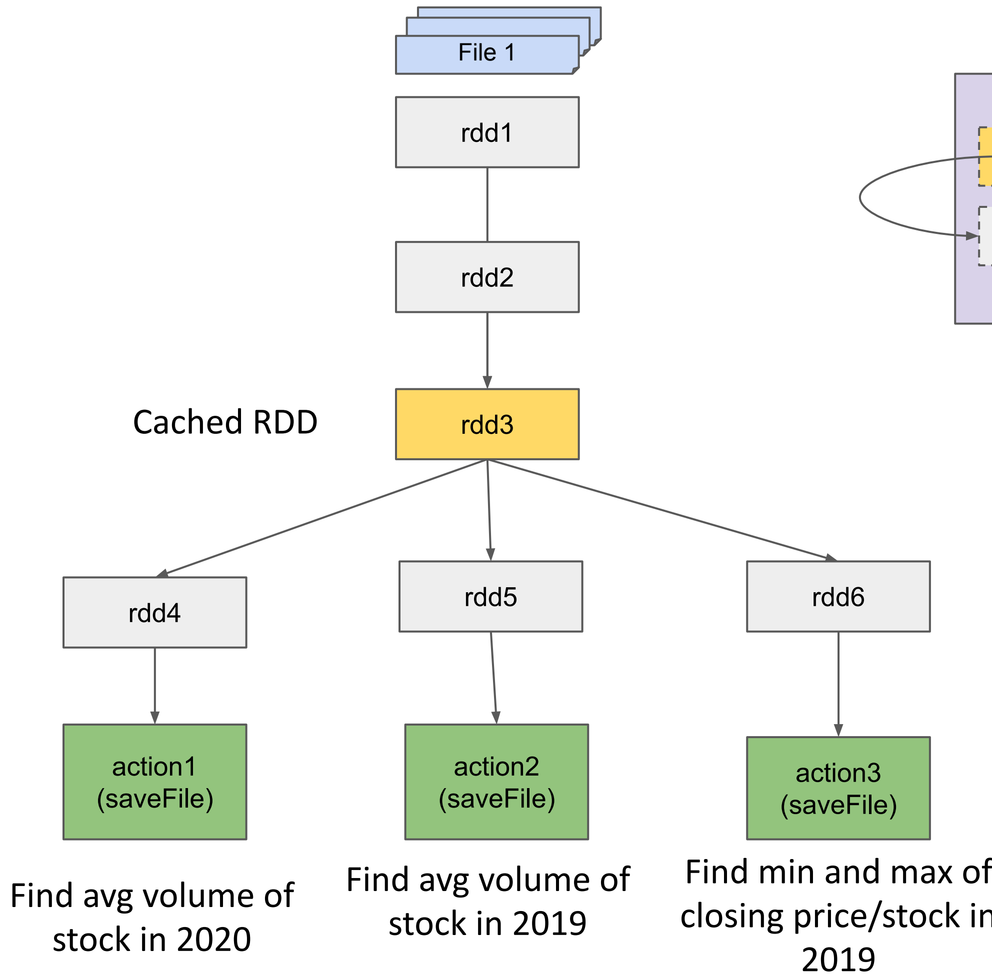
- createCombiner, which turns a V into a C (e.g., creates a one-element list)
- mergeValue, to merge a V into a C (e.g., adds it to the end of a list)
- mergeCombiners, to combine two C’s into a single one (e.g., merges the lists)

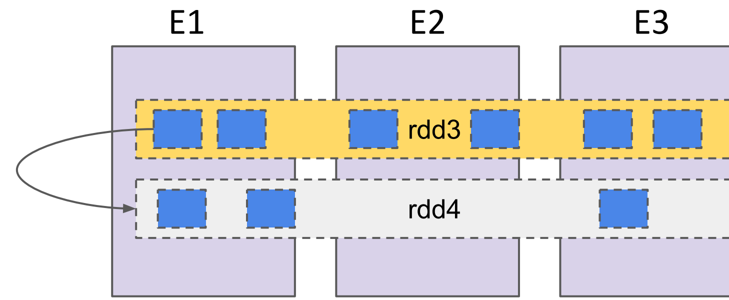
```
>>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 2)])
>>> def to_list(a):
...     return [a]
...
>>> def append(a, b):
...     a.append(b)
...     return a
...
>>> def extend(a, b):
...     a.extend(b)
...     return a
...
>>> sorted(x.combineByKey(to_list, append, extend).collect())
[('a', [1, 2]), ('b', [1])]
```

Scenario: executor core = 100, Total blocks = 8000 (1TB file)

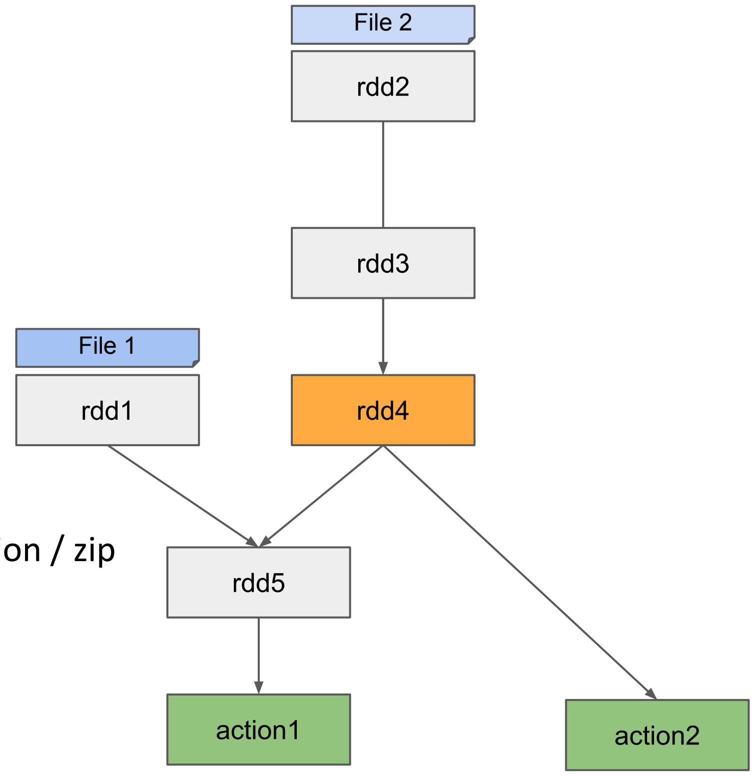
- 25 executors with 4-cores/ executor
- 8000 blocks will requires 8000 tasks (generally speaking)
- Spark will execute 100 tasks at a time in parallel. Once finished, it will pick next set of 100 tasks, it continues so on..

DAG - Directed Acyclic Graph





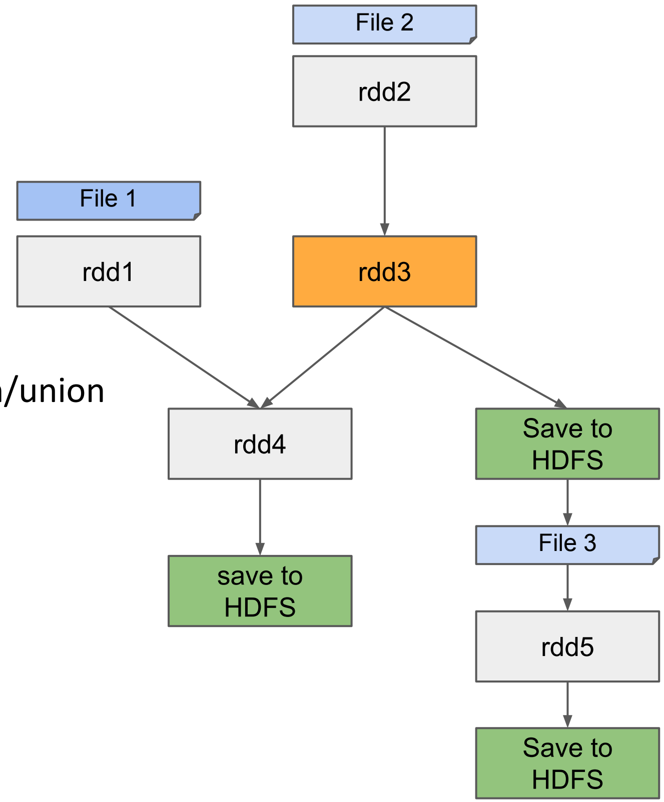
DAG



join / union / intersection / zip

DAG

Operations: join/union



RDD partitions - learning objectives

- Why partitions are important
  - controls parallelism of a spark job, hence speed
- Find number of partitions of an RDD
- What controls the number of partitions of a base RDD
- What controls number of partitions for narrow and wide transformation
- Change number of partitions:
  - increase, decrease, with shuffle of not
- Implications of partitions on output

RDD partitions

- Partitions are like shards of RDD - each partition ***contains*** multiple records
- A partition belongs to only one executor, but one executor can contain multiple partitions

How are the files are stored in HDFS

- When you store data on HDFS, the files are broken into blocks .. max block 128 MB by default
- Each file will contain at least one block
    - 1 GB one plain text file - 8 blocks, each approximately 128 MB
    - 1000 files each 1 MB - 1000 blocks, each 1 MB

Number of partitions in **base RDD**

HDFS data source -

- For splittable files (e.g. csv), the number of partitions = number of HDFS blocks
- For non-splittable files (e.g. .gz files), the number of partitions = number of files

S3, or NFS or local file system

- For splittable files, the number of partitions = number of blocks with approximate chunk size ~ 32 MB
- For non-splittable files, the number of partitions = number of files

You can specify the minPartitions argument in textFile function as a hint. [It is the minimum number of partitions of the RDD]

```
sc.textFile("stocks", minPartitions = 5) // Works for splittable file
```

Number of partitions for child (derived) RDD

IF the child RDD is created out of <u>narrow transformation</u>

- number of partitions of the child = number of partitions of the parent

IF the child RDD is created out of <u>wide transformation</u>

1. You can specify the number of partitions as an argument for the wide operation

   … **`pairRdd.groupByKey(numPartitions = 7)`**

2. Else If do not specify numPartitions, then "`spark.default.parallelism`" property specifies the number of partitions
3. Else, the number of partitions of the parent RDD

Explicitly control number of partitions in code

Four functions to control number of partitions

- **repartition**: use it to increase the number of partitions, this transformation is a wide operation, hence it will increase the number of stages for a job. It shuffles the data to create a almost same number of records in each partition. Hence, it will eliminated any data skew in the partitions of the parent rdd.
- **coalesce**: reduce the number of partitions; this is a narrow transformation. By default, it does not shuffle the data among partitions.
- **partitionBy**
- specify numPartitions in wide operation like groupByKey, reduceByKey, sortBy

Partition By Example

```
val  result =  raw
        .filter(line => !line.startsWith("date"))
        .filter(line => line.startsWith("2019"))
        .keyBy(line => line.substring(5, 7).toInt)
        .partitionBy(new Partitioner(){
          override def numPartitions: Int = 12
          override def getPartition(key: Any): Int = {
            (key.asInstanceOf[Int] -1) % numPartitions
          }
        })
        .map(t => t._2)
        .mapPartitions(partition =>  {
          partition
            .toList
            .sortBy(line => line.split(",")(7))
            .toIterator
        })
```

Number of partitions in DataFrame

`spark.sql.files.maxPartitionBytes (Default: 128 MB)` - The maximum number of bytes to pack into a single partition when reading files.

`spark.sql.files.openCostInBytes (Default – 4MB)` - The estimated cost to open a file, measured by the number of bytes could be scanned in the same time. This is used when putting multiple files into a partition. It is better to over-estimated, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first).

`spark.sql.shuffle.partitions (Default: 200)` - Configures the number of partitions to use when shuffling data for joins or aggregations.

Number of output files in dataframe write operations

Control the maximum number of records in a part file when dataframe in written

`spark.sql.files.maxrecordsperfile` - (works in AWS)

Use hadoop 2 output committer

```
spark.conf.set("spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version", "2")
```

RDD - Resilient Distributed Dataset

- RDD's are immutable, you can consider them as view in database
- An RDD can be transformed as another RDD
- You can imagine them distributed array that span across multiple JVM processes running on multiple servers in a cluster
- RDD consists of multiple partitions - one executors can contain multiple partitions, but a partition belongs to one executor only
- When a job is performed, spark processes each partitions independently and aggregates the final output

Word count program in RDD

Python

```
(sc.textFile("<path>")
.flatMap(lambda line: line.split(" "))
.map(lambda token: (token, 1))
.reduceByKey(lambda x, y: x+y)
.collect())
```

Scala

```
sc.textFile("<path>")
.flatMap(line => line.split(" "))
.map(token => (token, 1))
.reduceByKey((x, y) => x+y)
.collect()
```

RDD - broadcast and accumulator

- **Broadcast variables** allow the programmer to keep a read-only variable cached on each executor rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms (p2p) to reduce communication cost.
- **Accumulators** are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#shared-variables

Cache

- **Unless specified by the developer, spark does not cache any RDD / dataframe**
- Two functions - cache and persist to put rdd/dataframe into cache
    - Both cache and persist are lazy operation. Cache will be materialized when an action is performed on the rdd/dataframe
    - **cache** function is basic. **persist** function gives more control on caching behaviour such as how many replicas of caches we want, where to cache (memory vs disk), format of the cache (stores data serialized objects i.e. byes or deserialized objects i.e. java objects
    - _SER type cache levels have no effect when using pyspark. In pyspark, the cached data reside python kernel as "pickle" objects, which are serialized by nature
- To remove the dataset from cache - use unpersist function on the RDD or DataFrame object
- Rule of thumb: 50% of executor memory is available for caching
- Caching is based on LRU scheme - least recently used cache will be evicted if new cache request do not have available resource
- Cached data are automatically removed after the session gets over

Which RDD/dataset is good candidates for caching?

- after loading a dataset from external datasource like RDBMS, S3, Nosql databases
- resulting RDD out of a filtering/sampling operations,
    - also you should consider coalescing the data
- after a wide operations like - repartition, groupByKey, sort, or join
- dataset used for a frequent subsequent actions like queries or ML training

Cache options

_SER levels are applicable in only in java/scala and RDD

Storage Level	Meaning
MEMORY_ONLY (default)	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. In pyspark the cached data stored in python VM in pickle format. **This is the default level.**
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2 , MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Un-cache

- You can call unpersist method on the RDD to un-cache it. It is an eager operation
- If the cache of the spark cluster is full, and you try to cache another RDD, Spark will evict the LRU schema to evict least used RDD

Build and Deploy Application

- Scala project - https://github.com/abulbasar/SparkScalaExamples
- Java project - https://github.com/abulbasar/SparkJavaExamples

- Install scala plugin if required
- Configure scala plugin with version 2.11
- Create a new project with maven support
- Add framework support for Scala
- Create a scala source directory src/main/scala
- Mark src/main/scala as source folder
- To build scala project using maven, add scala plugin in pom.xml [Refer the scala project in the link above]

Check JDK version of YARN processes

```
$ sudo jps -l
<grab the pid of the node manager>

$ sudo lsof -p <pid>| grep  -i jdk
java    6138 yarn  txt    REG              253,0     6400  544987
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/bin/java
java    6138 yarn  mem    REG              253,0     6144  545013
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/amd64/libjaas_unix.so
java    6138 yarn  mem    REG              253,0   306223  545057
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/ext/sunjce_provider.jar
java    6138 yarn  mem    REG              253,0    47400  545029
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/amd64/libsunec.so
java    6138 yarn  mem    REG              253,0    51761  545056
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/ext/sunec.jar
java    6138 yarn  mem    REG              253,0   280125  545058
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/ext/sunpkcs11.jar
java    6138 yarn  mem    REG              253,0   809216  545074
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/jsse.jar
java    6138 yarn  mem    REG              253,0    39320  545022
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/amd64/libmanagement.so
java    6138 yarn  mem    REG              253,0   101336  545024
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.252.b09-2.el6_10.x86_64/jre/lib/amd64/libnet.so
...
```

Submit scala/java application to YARN cluster

```
Set JAVA_HOME and SPARK_HOME in /etc/hadoop/conf/hadoop-env.sh
Set HADOOP_CONF_DIR in /etc/spark/conf/spark-env.sh

sudo /etc/init.d/hadoop-yarn-nodemanager restart
sudo /etc/init.d/hadoop-yarn-resourcemanager restart
```

```
$ spark-submit \
    --class org.apache.spark.examples.SparkPi \
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 4g \
    --executor-memory 16g \
    --executor-cores 4 \
    --queue default \
    --num-executors 10 \
    --conf spark.default.parallelism=16 \
    spark-examples*.jar \
    10
```

Pyspark

```
$ spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 4g \
    --executor-memory 16g \
    --executor-cores 4 \
    --queue default \
    --num-executors 10 \
    --conf spark.default.parallelism=16 \
    --py-files <list of .py modules> \
    <python_script.py> \
    10
```

Spark Configurations

- Linux ENVIRONMENT variables
  - Mention the ENV variables in $SPARK_HOME/conf/spark-env.sh. You need to copy this file at each of the nodes
  - The ENV variables which are only applicable to Driver, can be set in the terminal prior to launching spark-submit command
- Spark Configuration properties
  - At code you set/setIfMissing method on the SparkConf object
  - In the spark-submit, pass the properties using --conf argument
  - Set the properties in the $SPARK_HOME/conf/spark-defaults.conf

All available properties

https://spark.apache.org/docs/latest/configuration.html

https://spark.apache.org/docs/latest/monitoring.html

Deployment mode

Cluster

- Driver runs inside the App Master (YARN)
- If the spark-submit is terminated while the application is still running, the application keeps running inside the cluster. If you want to kill the background application, use "yarn application -kill" command.
- Suitable for running batch jobs, streaming jobs

Client

- Driver runs on the edge node on which spark-submit is called
- If the spark-submit is terminated while the application is running, the application will be terminated too.
- Suitable use cases - interactive shells (REPL), notebooks

Pass additional libraries in spark-submit

```
libs="/absolute/path/to/libs/*"

spark-submit \
  ...
  --master yarn \
  --conf "spark.driver.extraClassPath=$libs" \
  --conf "spark.executor.extraClassPath=$libs" \
  ...
  /my/application/application-fat.jar \
  param1 param2
```

Submit pyspark application to YARN cluster

```
$ export HADOOP_CONF_DIR=XXX
$ ./bin/spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --driver-memory 4g \
    --executor-memory 16g \
    --executor-cores 4 \
    --queue default \
    --num-executors 4 \
    --py-files <required python files comma separated paths> \
    word-count.py \
    [args …]
```

Log4j Properties for controlling logging level

If required, copy the file $SPARK_HOME/conf/log4j.properties.template as $SPARK_HOME/conf/log4j.properties.

Set root logging level to WARN and resubmit the job. You should less logging output messages.

```
# Set everything to be logged to the console
log4j.rootCategory=WARN, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n
```

Supply external libraries while submitting spark job

1. You can specify --packages. It requires access to maven central or company hosted maven repo. For example,

   $ spark-submit --packages mysql:mysql-connector-java:8.0.15 <project.jar> ...

2. Supply jars using --jars

   $ spark-submit --jars mysql_mysql-connector-java-8.0.15.jar,com.google.protobuf_protobuf-java-3.6.1.jar <project.jar>

3. Create a fat jar. Fat jar includes all dependent jars along with your own jar.

   $ spark-submit <project.jar>

Quiz

1. Which returns all records of RDD to the driver?
   a. collect
   b. take
   c. map
2. Classify the operations as transformation or action
   a. collect
   b. map
   c. foreach
   d. reduce
   e. distinct
   f. sample
3. In a spark session, how many jobs can run in parallel?
   a. Only one
   b. More than one
4. Within a stage, the tasks run in __?
   a. sequential order
   b. parallel

Checkpointing

- Checkpointing is mainly used in iterative algorithms and Streaming processes.
- On batch processing we are used to having fault tolerance(caching or persisting). This means, in case a node crashed, the job doesn't lose its state and the lost tasks are rescheduled on other workers. Intermediate results are written to persistent storage(that has to be fault tolerant as well like HDFS, or Cloud Object Storage)
- Maintaining RDD lineage(caching or persisting) provides resilience but can also cause problems when the lineage gets very long - For example: iterative algorithms, streaming - Recovery can be very expensive - Potential stack overflow
- Checkpointing saves the data to HDFS - Provides fault-tolerant storage across nodes - Lineage is not saved - Must be checkpointed before any actions on the RDD

Checkpoint

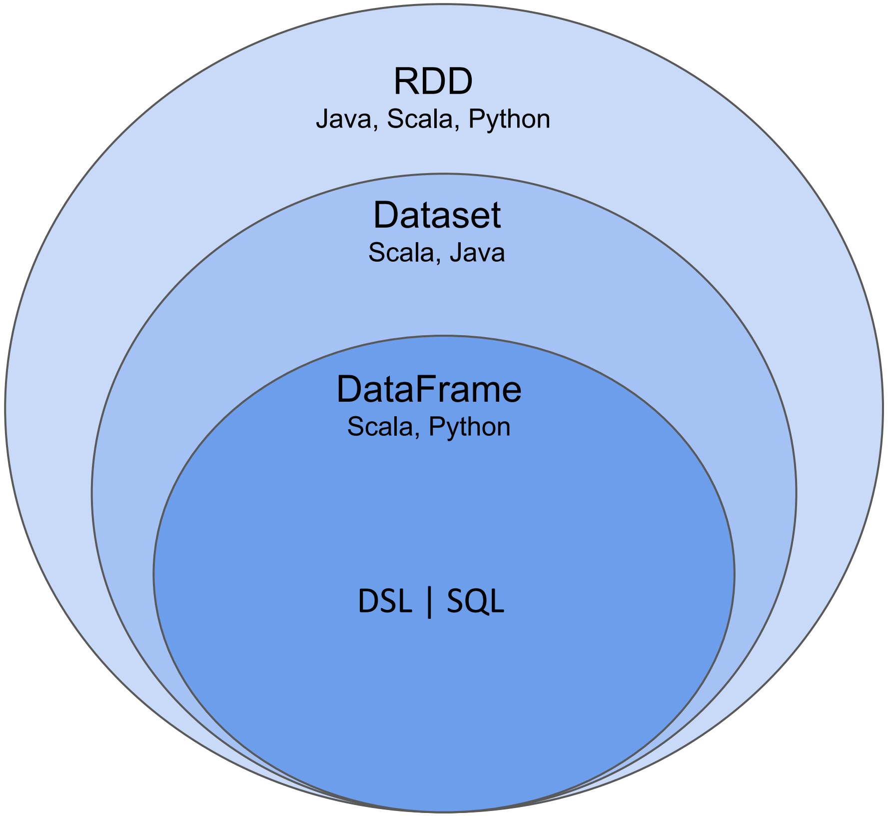
- Using Dataset checkpointing requires that you specify the checkpoint directory.
- The directory stores the checkpoint files for RDDs to be checkpointed.
- Use SparkContext.setCheckpointDir to set the path to a checkpoint directory.
- Checkpointing can be local or reliable which defines how reliable the checkpoint directory is.
- Local checkpointing uses executor storage to write checkpoint files to and due to the executor lifecycle is considered unreliable.
- Reliable checkpointing uses a reliable data storage like Hadoop HDFS.

DataFrame

DataFrame/Dataset

- Spark DataFrame is similar to DataFrame in Python Pandas and R
- DataFrame is like a view in RDBMS - it contains named columns and each column has a specific type - string, double, int, boolean, **list, set, struct, Row*** etc.
- DataFrame is RDD[Row], (pyspark.sql.Row, org.apache.spark.sql.Row)
- If you have have a dataframe, you can register a view ... You **cannot** do any mutation operations (insert/update/delete) but rather only SELECT queries are allowed.
- DataFrame supports 3 kinds of operations - DSL and SQL operations are equally performant
  - DataFrame DSL (Api)
  - Spark SQL - ANSI SQL support
  - RDD operations
- Supports analytical functions (window operations, rollups, cubes, pivots) and statistical functions
- Support wide range of UDF (user defined functions). You can define your own.
- DataFrame output can be written to HDFS, S3, RDBMS, NoSQL databases etc.

RDD vs Dataset vs DataFrame



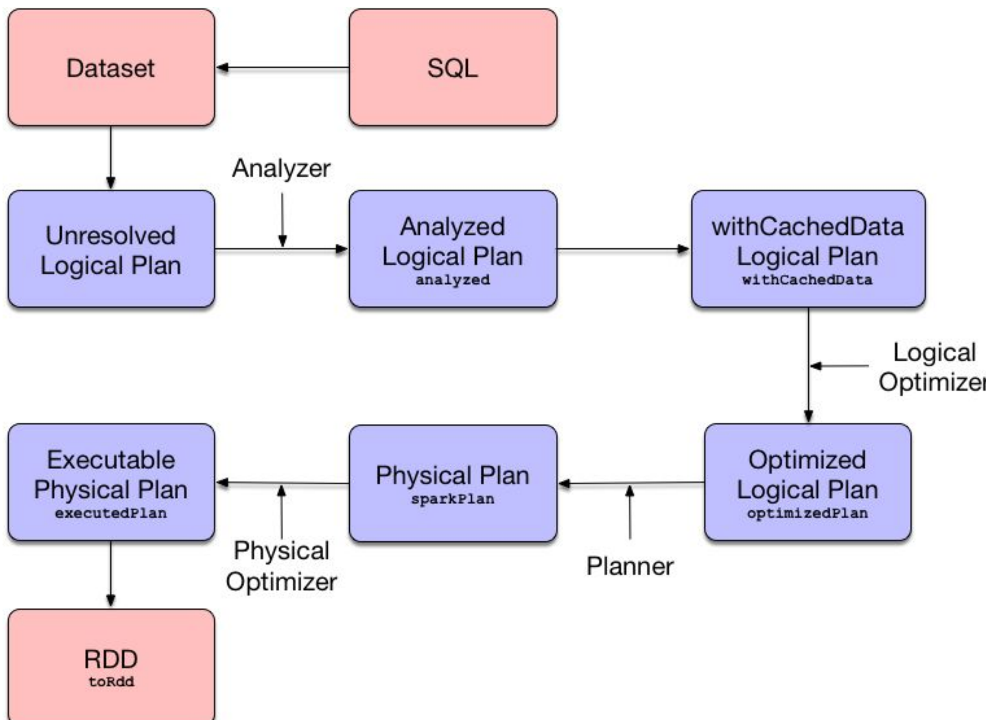
RDD: Collection of potentially heterogeneous records

Dataset: Collection of homogeneous records (each has the same schema)

DataFrame: It is an alias for Dataset of "Row" class objects.

- Dataset is available only in Java and Scala
- DataFrame is available in Python and Scala
- All operations supported by DataSet are available in DataFrame
- All RDD operations are available on Dataset/DF
- Dataset gives compile time check data structure
- Both dataset and dataframe have typed columns

Execution Flow



https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-QueryExecution.html

DataFrame is an alias for Dataset[Row]

```scala
package org.apache.spark
package object sql extends scala.AnyRef {
  @org.apache.spark.annotation.InterfaceStability.Unstable
  @org.apache.spark.annotation.DeveloperApi
  type Strategy = org.apache.spark.sql.execution.SparkStrategy
  type DataFrame = org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
  private[sql] val SPARK_VERSION_METADATA_KEY : java.lang.String = { /* compiled code */ }
}
```

Typed (Dataset) vs Untyped (DataFrame) ops

```
case class DeviceData(device: String, deviceType: String, signal: Double, time: DateTime)

val df: DataFrame = ... // streaming DataFrame with IOT device data with schema { device: string, deviceType: st
ring, signal: double, time: string }
val ds: Dataset[DeviceData] = df.as[DeviceData]    // streaming Dataset with IOT device data

// Select the devices which have signal more than 10
df.select("device").where("signal > 10")      // using untyped APIs
ds.filter(_.signal > 10).map(_.device)         // using typed APIs

// Running count of the number of updates for each device type
df.groupBy("deviceType").count()                          // using untyped API

// Running average signal for each device type
import org.apache.spark.sql.expressions.scalalang.typed
ds.groupByKey(_.deviceType).agg(typed.avg(_.signal))    // using typed API
```

Key points:

- Typed expressions give compile time checks
- Untypes expressions give run time checks
- Performance is same

Two ways to create a DataFrame/Dataset

A. Create dataframe using spark packages

- **csv**
- **json**
- **parquet**
- **orc**
- jdbc
- avro
- xml
- nosql
- etc. https://spark-packages.org/

B. Create a dataframe by converting RDD

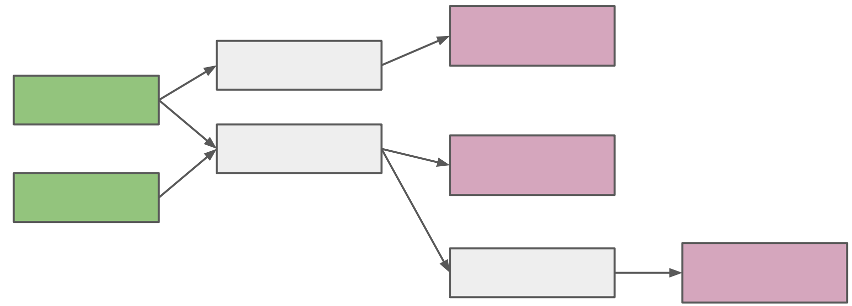
- Suitable when you need complex parsing logic

1. Create an RDD Or Dataset[String]
2. Custom parsing logic to create a homogeneous data structure
3. Convert the RDD of homogeneous records to DataFrame/Dataset

Maven dependency for avro, xml format in Spark Dataframe

```xml
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-avro_${scala.version}</artifactId>
    <version>${spark.version}</version>
</dependency>

<dependency>
    <groupId>com.databricks</groupId>
    <artifactId>spark-xml_${scala.version}</artifactId>
    <version>0.14.0</version>
</dependency>
```

DataFrame pipeline



- source system: hdfs, s3
- file formats: csv, xml, parquet
- JDBC
- NoSQL databases
- Kafka stream

- create/modify/drop columns
- join, union etc multiple data frames
- sort
- aggregation
- filter

- show the output to console
- write to hive table
- save to HDFS/S3
- write to kafka stream

DataFrame writer

- Choose the file format - csv, json, xml, avro, parquet, orc etc.
- Control compression - whether to compress, compression codec
- Number of files by controlling number of partitions
- Partition By
- Overwrite/append/error mode
- save as Hive table - persist the data and schema
- Save dataframe to nosql/jdbc etc.

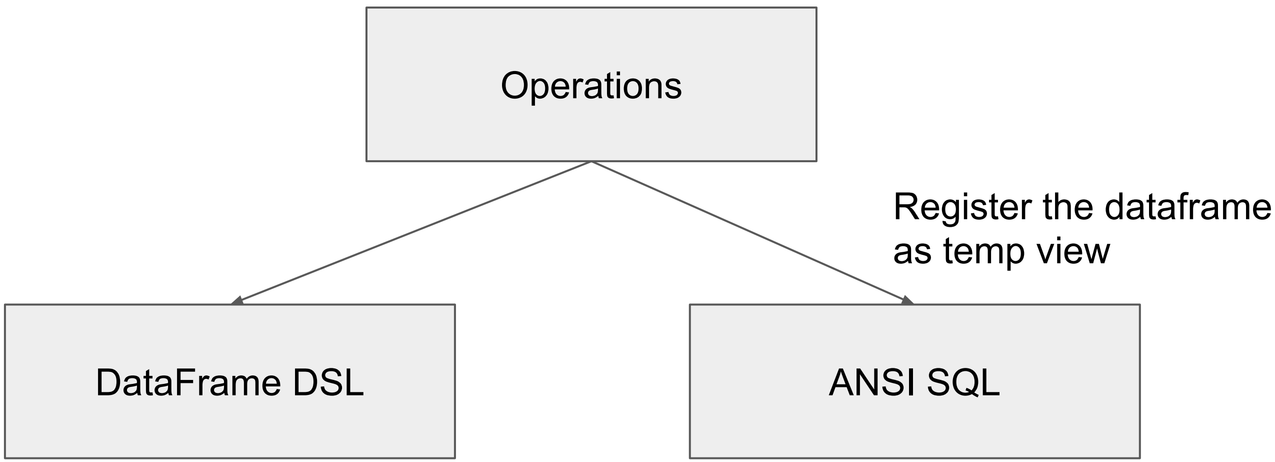
Dataframe transformation operations

- Add/drop/rename columns
- Filter rows of the dataframe - one or more conditions
- Group by - one or more fields
- Aggregate fields
    - aggregate grouped dataset (output of group by)
    - global aggregation
- Drop duplicates
- Distinct values (based on the hash of the entire row)
- Order By
- Custom udf
- Rollup, cube, pivot
- Double Rdd functions
- Window operation - lag, lid, row_num, avg, stddev ..
- Rolling aggregation for time series

Group a dataset by multiple fields

```
private RelationalGroupedDataset groupBy(Dataset<Row> input, String ... cols){
        List<Column> sparkCols = new ArrayList<>();
        for (String col : cols) {
            sparkCols.add(functions.col(col));
        }
        final Seq<Column> columnSeq =
JavaConverters.collectionAsScalaIterableConverter(sparkCols).asScala().toSeq();
        return input.groupBy(columnSeq);
    }
```

DataFrame operations




```
stocks
.filter("year(date)=2018")
.groupBy("symbol")
.agg(expr("avg(volume)").alias("avg_vol"))
```

```
stocks.createOrReplaceTempView("stocks")

spark.sql("select symbol, avg(volume)
avg_vol from stocks where year(date) =
2018 group by symbol")
```

- Performance wise both options are equal
- In addition to using Dataframe API and SQL operation, you can also use RDD operations

Convert list to dataframe

```
df = spark.createDataFrame([
    ("Mercury", 57.9,87.96, 100),
    ("Venus", 108.2, 224.68, 726),
    ("Earth", 149.6, 365.26, 260),
    ("Mars", 227.9, 686.98, 150),
    ("Jupyter", 778.3, 11.862, 120),
    ("Saturn", 1427.0, 29.466, 88),
    ("Uranus", 2871.0, 84.07, 59),
    ("Neptune", 4497.1, 164.81, 48),
    ("Pluto", 5913.0, 247.7, 37),
], ["planel", "dis_mnK", "period", "temp_low" ])
```

```
+-------+-------+------+--------+
| planel|dis_mnK|period|temp_low|
+-------+-------+------+--------+
|Mercury|   57.9| 87.96|     100|
|  Venus|  108.2|224.68|     726|
|  Earth|  149.6|365.26|     260|
|   Mars|  227.9|686.98|     150|
|Jupyter|  778.3|11.862|     120|
| Saturn| 1427.0|29.466|      88|
| Uranus| 2871.0| 84.07|      59|
|Neptune| 4497.1|164.81|      48|
|  Pluto| 5913.0| 247.7|      37|
+-------+-------+------+--------+
```

Word count program in DataFrame

```
from pyspark.sql import functions as F

(spark
.read
.text("<file>")
.select(F.explode(F.expr("split(value, ' ')")).alias("word"))
.groupBy("word")
.count()
.show())
```

RDD vs DataFrame

- While using RDD, you have to parse the data, but in dataframe you specify format like csv, and spark is going to parse the data for you
- RDD supports functional programming API patterns for writing the logic, while dataframe uses SQL or Dataframe DSL
- Dataframe ops are optimized by catalyst + DAG optimizers, RDD operations do not use catalyst
- Dataframe operations are executed with Tungsten (it is a memory management system within spark). RDD do not use Tungsten.

Advantages of Dataframe over RDD

- Performance of spark application on DataFrame is almost same in all four programming languages - scala, java, python, R. For RDD operations java and scala performs equally, while Python is significantly slower.
- Dataframe supports various data formats like csv, json, parquet out of the box, so no parsing is required
- Dataframes provide DSL, SQL operations - this makes faster development
- Dataframe operations are optimized by Catalyst out of the box
- DataFrame operations run on Tungsten engine which operates on Serialized format of the data making operations cpu, memory optimized.

Options for csv reader

https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/DataFrameReader.html#csv(java.lang.String...)

csv

`public Dataset<Row> csv(String... paths)`

Loads a CSV file and returns the result as a `DataFrame`.

This function will go through the input once to determine the input schema if `inferSchema` is enabled. To avoid going through the entire data once, disable `inferSchema` option or specify the schema explicitly using `schema`.

You can set the following CSV-specific options to deal with CSV files:

- `sep` (default `,`): sets the single character as a separator for each field and value.
- `encoding` (default `UTF-8`): decodes the CSV files by the given encoding type.
- `quote` (default `"`): sets the single character used for escaping quoted values where the separator can be part of the value. If you would like to turn off quotations, you need to set not `null` but an empty string. This behaviour is different form `com.databricks.spark.csv`.
- `escape` (default `\`): sets the single character used for escaping quotes inside an already quoted value.
- `comment` (default empty string): sets the single character used for skipping lines beginning with this character. By default, it is disabled.
- `header` (default `false`): uses the first line as names of columns.
- `inferSchema` (default `false`): infers the input schema automatically from data. It requires one extra pass over the data.
- `ignoreLeadingWhiteSpace` (default `false`): defines whether or not leading whitespaces from values being read should be skipped.
- `ignoreTrailingWhiteSpace` (default `false`): defines whether or not trailing whitespaces from values being read should be skipped.
- `nullValue` (default empty string): sets the string representation of a null value. Since 2.0.1, this applies to all supported types including the string type.
- `nanValue` (default `NaN`): sets the string representation of a non-number" value.
- `positiveInf` (default `Inf`): sets the string representation of a positive infinity value.
- `negativeInf` (default `-Inf`): sets the string representation of a negative infinity value.
- `dateFormat` (default `yyyy-MM-dd`): sets the string that indicates a date format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to date type.
- `timestampFormat` (default `yyyy-MM-dd'T'HH:mm:ss.SSSZZ`): sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to timestamp type. `java.sql.Timestamp.valueOf()` and `java.sql.Date.valueOf()` or ISO 8601 format.
- `maxColumns` (default `20480`): defines a hard limit of how many columns a record can have.
- `maxCharsPerColumn` (default `1000000`): defines the maximum number of characters allowed for any given value being read.
- `maxMalformedLogPerPartition` (default `10`): sets the maximum number of malformed rows Spark will log for each partition. Malformed records beyond this number will be ignored.
- `mode` (default `PERMISSIVE`): allows a mode for dealing with corrupt records during parsing.
  - `PERMISSIVE` : sets other fields to `null` when it meets a corrupted record. When a schema is set by user, it sets `null` for extra fields.
  - `DROPMALFORMED` : ignores the whole corrupted records.
  - `FAILFAST` : throws an exception when it meets corrupted records.

https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/DataFrameReader.html#csv-scala.collection.Seq-

https://spark.apache.org/docs/latest/sql-data-sources-csv.html

Reader: DataFrame from RDBMS

For mysql driver support, start the program with mysql jdbc driver. For REPL, for example:

```
$ spark-shell --packages mysql:mysql-connector-java:5.1.47

(spark
    .read
    .format("jdbc")
    .option("url", "jdbc:mysql://127.0.0.1/retail_db")
    .option("driver", "com.mysql.jdbc.Driver")
    .option("dbtable", "(select * from orders where order_id < 20) as orders")
    .option("user", "root")
    .option("password", "root")
    .option("lowerBound", 1)
    .option("upperBound", 20)
    .option("partitionColumn", "order_id")
    .option("numPartitions", 3)
    .load()).show()
```

Can be name of a DB table as well in place of this sql statement

Reading from mongodb

Reader: DataFrame from Cassandra Table

Pyspark

```
(spark.
 read.
 format("org.apache.spark.sql.cassandra").
 options(table = "ratings", keyspace = "lab").
 load())
```

Spark SQL Guide

https://docs.databricks.com/spark/latest/spark-sql/index.html

- Cost-Based Optimizer
- Transforming Complex Data Types
- Data Skipping Index
- Transactional Writes to Cloud Storage with DBIO
- Handling Bad Records and Files
- Higher Order Functions
- Task Preemption for High Concurrency
- Handling Large Queries in Interactive Workflows
- Skew Join Optimization
- Structured Data Access Controls
- User Defined Aggregate Functions - Scala
- User Defined Functions - Python
- User Defined Functions - Scala

Spark SQL Tuning Guide

https://spark.apache.org/docs/latest/sql-performance-tuning.html

- Caching Data In Memory
- Other Configuration Options
- Join Strategy Hints for SQL Queries
- Coalesce Hints for SQL Queries
- Adaptive Query Execution

Calculate table statistics

```
// example 1
val res = spark.sql("ANALYZE TABLE mytablename COMPUTE STATISTICS FOR COLUMNS col_name1, col_name2")

// example 2
val res = spark.sql("ANALYZE TABLE mytablename COMPUTE STATISTICS FOR COLUMNS col_name1,
col_name2").queryExecution.logical
import org.apache.spark.sql.execution.command.AnalyzeColumnCommand

// example 3
val res = spark.sql("ANALYZE TABLE mytablename COMPUTE STATISTICS FOR ALL COLUMNS")

// example 4
val res = spark.sql("ANALYZE TABLE mytablename COMPUTE STATISTICS FOR COLUMNS col_not_exists")
```

spark.sql("show table stats mytablename")

https://issues.apache.org/jira/browse/SPARK-25458

Which column statistics are collected

`spark.sql.statistics.histogram.enabled` is off by default.

Name	Description
distinctCount	Number of distinct values
min	Minimum value
max	Maximum value
nullCount	Number of null values
avgLen	Average length of the values
maxLen	Maximum length of the values
histogram	Histogram of values (as Histogram which is empty by default)

https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-ColumnStat.html

Hive and Spark

- Hive a popular Hadoop ecosystem tool
  - Its popularity lies in the fact that it support SQL queries
  - Suitable for interactive analytics and ETL
  - Fault Tolerant, rich set of UDF's
  - Not designed for low latency and high concurrency
  - Supports HDFS and HBase as data layer
- Modes
  - Hive on Spark - we can configure Hive to run HQL using Spark's execution engine. Users interact with standard hive interface.
  - Use Hive's metastore within Spark - metastore is the persisted schema definition. Note: in spark 2.x Hive context and SQLContext merged into Spark Session.

Configurations for hive

Create symlink or copy the following config files inside $SPARK_HOME/conf

core-site.xml -> /etc/hadoop/conf/core-site.xml
hdfs-site.xml -> /etc/hadoop/conf/hdfs-site.xml
hive-site.xml -> /etc/hive/conf/hive-site.xml

```
final SparkConf sparkConf = new SparkConf()
        .setAppName(getClass().getName())
        .setIfMissing("spark.master", "local[*]");

final SparkSession session = SparkSession
        .builder()
        .enableHiveSupport()
        .config(sparkConf).getOrCreate();
```

Test Hive Metastore

```
$ mysql -h 127.0.0.1 -uroot -ptraining

mysql> create database hivemetastore;
mysql> ALTER DATABASE  hivemetastore CHARACTER SET latin1;
```

Test hive metastore connection

```
$ cd $SPARK_HOME
$ bin/run-example sql.hive.SparkHiveExample
```

Source code

```
mysql> select * from DBS;
```

```
$ hive --hiveconf hive.root.logger=INFO,console -e "select count(1) from stocks;"
```

Demonstration for Hive integration for Spark

1. Upload movies.csv to HDFS /user/cloudera/movielens/movies
2. Add hive configuration to spark conf and launch spark application
3. Define a table in hive (movies) and check whether you can query the table in hive
4. Check whether the table (movies) is visible in spark and you can query in spark
5. Upload stocks.small.csv to HDFS path /user/cloudera/stocks
6. Create a dataframe in Spark for the stocks data loaded in the previous step
7. Save the dataframe as table - stocks
8. Check whether the table (stocks) is visible in hive and you can query the table

Insert into hive partition

spark.conf.set("hive.exec.dynamic.partition", "true")

spark.conf.set("hive.exec.dynamic.partition.mode", "nonstrict")

```
part = (spark.read.load("stocks_parquet")
.withColumn("year", F.expr("year(date)"))
.withColumn("date", F.expr("cast(cast(date as date) as string)"))
.withColumnRenamed("date", "tr_date")
.where("year = 2016")
)

part.write.mode("append").insertInto("stocks_bucket_hive")
```

```
hive>
CREATE EXTERNAL TABLE `stocks_bucket`(
`tr_date` string,
`open` double,
`high` double,
`low` double,
`close` double,
`volume` double,
`adjclose` double,
`symbol` string)
PARTITIONED BY(year int)
CLUSTERED BY (`symbol`) INTO 4 BUCKETS;
```

Drop duplicate / distinct

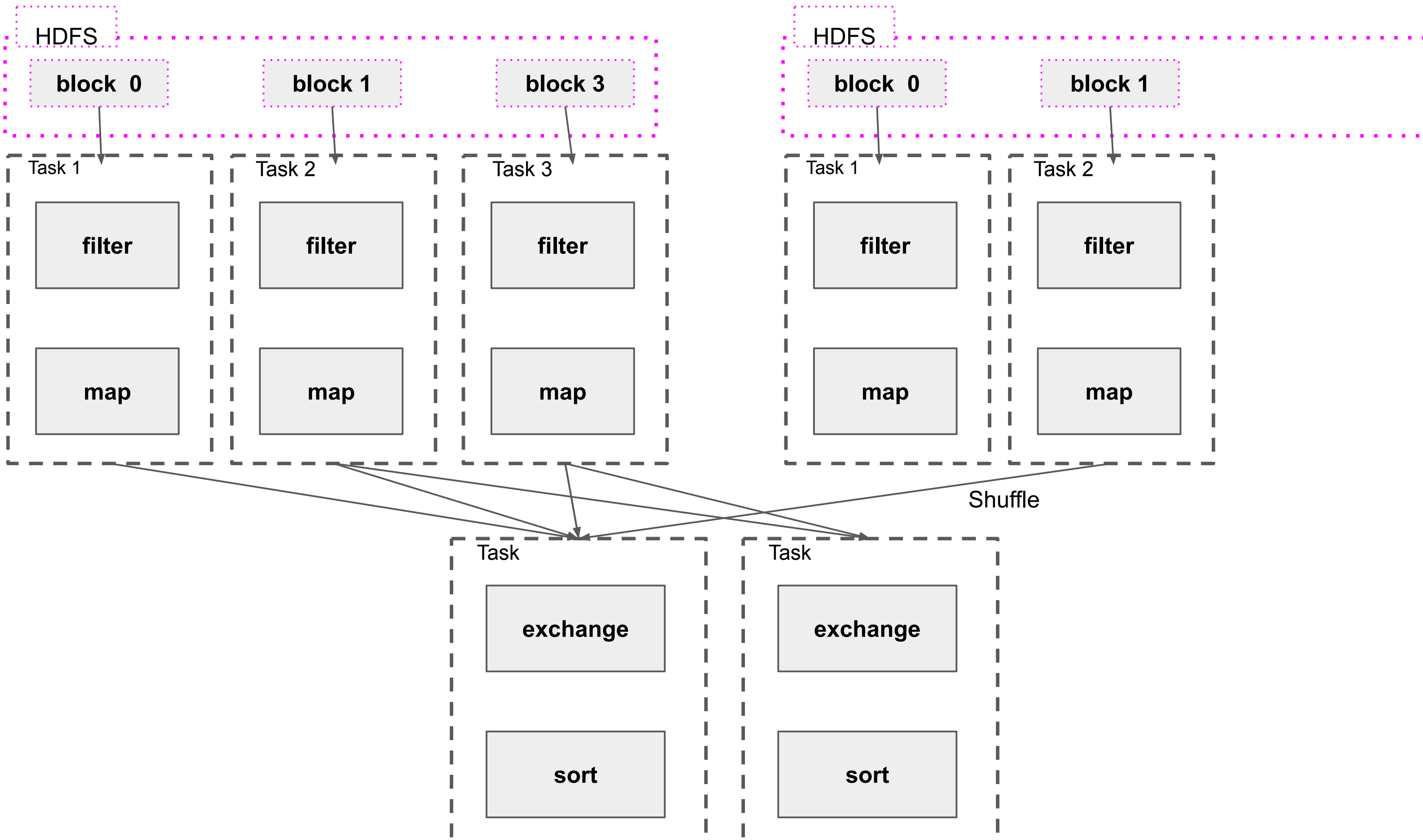
```
import spark.implicits._

val simpleData = Seq(("James", "Sales", 3000),
  ("Michael", "Sales", 4600),
  ("Robert", "Sales", 4100),
  ("Maria", "Finance", 3000),
  ("James", "Sales", 3000),
  ("Scott", "Finance", 3300),
  ("Jen", "Finance", 3900),
  ("Jeff", "Marketing", 3000),
  ("Kumar", "Marketing", 2000),
  ("Saif", "Sales", 4100)
)
val df = simpleData.toDF("employee_name", "department", "salary")
df.distinct() OR df.dropDuplicates()

df.dropDuplicates("department","salary") // Drop duplicate based on columns
```

Intersection

Returns a new DataFrame containing rows only in both this frame and another frame. This is equivalent to INTERSECT in SQL.

```
df1 = spark.createDataFrame([("Alex", 20), ("Bob", 30), ("Cathy", 40)], ["name", "age"])

df2 = spark.createDataFrame([("Alex", 20), ("Doge", 30), ("eric", 40)], ["name", "age"])

df1.intersect(df2)
```

Join



HDFS
block 0
block 1
block 3
Task 1
Task 2
Task 3
filter
filter
filter
map
map
map
HDFS
block 0
block 1
Task 1
Task 2
filter
filter
map
map
Shuffle
Task
exchange
sort
Task
exchange
sort

Spark Join Types

- Inner Join: Returns only the rows from both the dataframes that have matching values in both columns specified as the join keys.
- Left / Left Outer Join: Returns all the rows from the left dataframe and the matching rows from the right dataframe. If there are no matching values in the right dataframe, then it returns a null.
- Right / Right Outer Join: Returns all the rows from the right dataframe and the matching rows from the left dataframe. If there are no matching values in the left dataframe, then it returns a null.
- Outer / Full Join: Returns all the rows from both the dataframes, including the matching and non-matching rows. If there are no matching values, then the result will contain a NULL value in place of the missing data.

Spark Join Types

Cross Join: Returns all possible combinations of rows from both the dataframes. In other words, it takes every row from one dataframe and matches it with every row in the other dataframe. The result is a new dataframe with all possible combinations of the rows from the two input dataframes. A cross-join is used when we want to perform a full outer join but in a more computationally efficient manner. Cross joins are not recommended for large datasets as they can produce a very large number of records, leading to memory issues and poor performance.

Left Anti Join: A left anti join in Spark SQL is a type of left join operation that returns only the rows from the left dataframe that do not have matching values in the right dataframe. It is used to find the rows in one dataframe that do not have corresponding values in another dataframe.

Spark Join Types

Left Semi Join: A left semi join in Spark SQL is a type of join operation that returns only the columns from the left dataframe that have matching values in the right dataframe. It is used to find the values in one dataframe that have corresponding values in another dataframe.

Self Join: A self join in Spark SQL is a join operation in which a dataframe is joined with itself. It is used to compare the values within a single dataframe and return the rows that match specified criteria.

Join execution strategy

- Broadcast Hash Join
- Shuffle hash join
- Shuffle sort-merge join
- Cartesian Join
- Broadcast nested loop join

https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf

Broadcast Join

- When joining between one large table with a small table, broadcasting the small table to each executor speeds up the join operation by reducing the shuffle
- Broadcast table is cached at the executors, so it will not shuffle the table if the join is performed multiple times
- Spark performs the broadcast join by default if the smaller table size is < 10 MB (this threshold can be configured by `spark.sql.autoBroadcastJoinThreshold`)
- Using Spark UI, you can observe that the join does not create a new stage and hence no shuffle

Broadcast Join Example

```
val csvOptions = Map("header" -> "true", "inferSchema" -> "true" )
val sp500 = spark.read.options(csvOptions).csv("/data/SP500.csv")
val stocks = spark.read.options(csvOptions).csv("/data/stocks.csv")
val resultWithBroadCast = stocks.join(F.broadcast(sp500), usingColumn =
"symbol")
```

```
Project [symbol#70, date#63, open#64, high#65, low#66, close#67, volume#68, adjclose#69, Security#13, SEC filings#14, GICS Sector#15, GICS
Sub Industry#16, Location#17, Date first added[3][4]#18, CIK#19, Founded#20]
+- *BroadcastHashJoin [symbol#70], [symbol#12], Inner, BuildRight
   :- *Project [date#63, open#64, high#65, low#66, close#67, volume#68, adjclose#69, symbol#70]
   :  +- *Filter isnotnull(symbol#70)
   :     +- *FileScan csv [date#63,open#64,high#65,low#66,close#67,volume#68,adjclose#69,symbol#70] Batched: false, Format: CSV, Location:
InMemoryFileIndex[file:/data/stocks.csv], PartitionFilters: [], PushedFilters: [IsNotNull(symbol)], ReadSchema:
struct<date:timestamp,open:double,high:double,low:double,close:double,volume:double,adjclose:doub...
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
      +- *Project [symbol#12, Security#13, SEC filings#14, GICS Sector#15, GICS Sub Industry#16, Location#17, Date first added[3][4]#18,
CIK#19, Founded#20]
         +- *Filter isnotnull(symbol#12)
            +- *FileScan csv [symbol#12,Security#13,SEC filings#14,GICS Sector#15,GICS Sub Industry#16,Location#17,Date first
added[3][4]#18,CIK#19,Founded#20] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/data/SP500.csv], PartitionFilters: [],
PushedFilters: [IsNotNull(symbol)], ReadSchema: struct<symbol:string,Security:string,SEC filings:string,GICS Sector:string,GICS Sub
Industry:stri...
```

Broadcast Join (a.k.a. map side join) in SQL

```
stocks.createOrReplaceTempView("stocks")
sp500.createOrReplaceTempView("sp500")
sql("select /*+ MAPJOIN(t2) */ * from stocks t1 join sp500 t2 on t1.symbol =
t2.symbol")
```

Other SQL hints

```
SELECT /*+ COALESCE(3) */ * FROM t
SELECT /*+ REPARTITION(3) */ * FROM t
SELECT /*+ REPARTITION(c) */ * FROM t
SELECT /*+ REPARTITION(3, c) */ * FROM t
SELECT /*+ REPARTITION */ * FROM t
SELECT /*+ REPARTITION_BY_RANGE(c) */ * FROM t
SELECT /*+ REPARTITION_BY_RANGE(3, c) */ * FROM t
SELECT /*+ REBALANCE */ * FROM t
SELECT /*+ REBALANCE(c) */ * FROM t
```

https://spark.apache.org/docs/latest/sql-ref-syntax-qry-select-hints.html#partitioning-hints

User defined function (UDF)

- We can define functions in Java, Scala, Python
- If you write an UDF in python, the UDF will be executed in Python kernel, that will slow down the dataframe operations. So, for time sensitive applications, you write the UDF in scala or Java, and call it in pyspark.

UDF in Scala

```
import org.apache.spark.sql.{functions => F}
spark.udf.register("area",(r:Int) => 3.14 * r * r)
spark.range(10).withColumn("area", F.expr("area(id)"))
```

This function create a dataframe with column with id

UDF in Python

```
import pyspark.sql.functions as F
spark.udf.register("area",lambda r: 3.14 * r * r)
spark.range(10).withColumn("area", F.expr("area(id)"))
```

This function create a dataframe with
column with id

Other UDF syntaxes in Pyspark

Annotation style

```python
@pandas_udf('long', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    # `v` is a pandas Series
    return v + 1  # outputs a pandas Series
```

```python
@pandas_udf('long', PandasUDFType.SCALAR_ITER)
def pandas_plus_one(itr):
    # `iterator` is an iterator of pandas Series.
    return map(lambda v: v + 1, itr)  # outputs an iterator
```

```python
@pandas_udf("id long", PandasUDFType.GROUPED_MAP)
def pandas_plus_one(pdf):
    # `pdf` is a pandas DataFrame
    return pdf + 1  # outputs a pandas DataFrame
```

```python
spark.range(10).select(pandas_plus_one("id"))
```

Project Zen (Spark 3.0) Style

```python
def pandas_plus_one(v: pd.Series) -> pd.Series:
    return v + 1
```

```python
def pandas_plus_one(itr: Iterator[pd.Series]) -> Iterator[pd.Series]:
    return map(lambda v: v + 1, itr)
```

```python
def pandas_plus_one(pdf: pd.DataFrame) -> pd.DataFrame:
    return pdf + 1
```

UDF Example

```
import json
from pyspark.sql.types import StructType, StructField, StringType, DoubleType, IntegerType, LongType

s = """{"id": "533c070a-68fc-11ee-9082-d10b03c656a7", "merchant_id": "m820", "customer_id": "c606163", "amount":
97.49700145035185, "category": "net", "timestamp": 1697114343255}"""
json.loads(s)

schema = StructType([
    StructField("id", StringType()),
    StructField("merchant_id", StringType()),
    StructField("customer_id", StringType()),
    StructField("amount", DoubleType()),
    StructField("category", StringType()),
    StructField("timestamp", LongType())
])

df = spark.createDataFrame([{"value": s}])
df = df.withColumn("value", F.from_json("value", schema))
df.selectExpr("value.*").printSchema()
```

Pivot

```
(df.selectExpr("*", "year(date) year", "month(date) month")
 .groupBy("year")
 .pivot("month")
 .agg(F.min("volume"))
).show()
```

year	1	2	3	4	5	6	7	8	9	10	11	12
2003	8988	8132	8988	8988	8988	8988	9416	8988	8988	9864	8151	9459
2007	9100	8645	10010	9100	10030	9600	9618	10555	8721	10585	9681	9220
2018	9558	8645	9637	9662	10120	9659	9660	10584	8747	10580	9659	8716
2015	9860	9367	10846	10353	9860	10851	10930	10437	10437	10944	9969	10977
2006	8924	8509	10317	8531	9882	9900	9009	10373	9038	9966	9522	9080
2013	10185	9215	9700	10670	10670	9726	10736	10736	9760	11224	9769	10269
2014	10269	9291	10269	10278	10290	10290	10781	10311	10316	11316	9348	10824
2019	9618	8694	9672	9729	10229	9184	10056	10058	9146	10514	9113	9466
2004	8600	8186	9913	9051	8620	9071	9093	9556	9156	9157	9177	9614
2020	1347	null	null	null	null	null	null	null	null	null	null	null
2012	9640	9640	10604	9653	10635	10164	10164	11132	9196	10164	10164	9680
2009	9340	8885	10296	9828	9360	10296	10296	9846	9849	10336	9411	10362
2016	9498	9998	10999	10499	10500	11016	10020	10959	9534	9534	9534	9534
2001	8580	7771	8998	8180	8998	8643	8708	9568	6240	9577	8780	8424
2005	8740	8303	9614	9177	9177	9614	8763	10112	9256	9282	9300	9362
2000	null	null	null	null	null	null	4423	9284	8080	8916	8526	8155
2010	8949	8949	10833	9894	9440	10384	9912	10384	9912	9912	9920	10406
2011	9463	9017	10941	9520	9996	10477	9540	10971	10028	10051	10068	10108
2008	9681	9220	9238	10192	9761	9778	10252	9786	9795	10741	8873	10274
2017	9080	8626	10442	8625	9988	9988	9080	10442	9080	9996	9555	9100

Rollup

```
df = spark.read.format("csv").options(header=True, inferSchema = True).load("stocks")
(df.selectExpr("*", "year(date) year", "month(date) month")
.rollup("symbol", "year", "month").agg(F.round(F.avg("volume"), 2))
.orderBy("year", "month")
.where("symbol like 'FB'")
)
```

```
+------+----+-----+--------------------+
|symbol|year|month|round(avg(volume), 2)|
+------+----+-----+--------------------+
|    FB|null| null|       4.394731818E7|
|    FB|2012| null|       5.464814645E7|
|    FB|2012|    5|      1.3905702222E8|
|    FB|2012|    6|        3.18052619E7|
|    FB|2012|    7|        2.47709381E7|
|    FB|2012|    8|       5.008456087E7|
|    FB|2012|    9|       5.571808947E7|
|    FB|2012|   10|       5.242563333E7|
|    FB|2012|   11|       7.273762857E7|
|    FB|2012|   12|          5.959161E7|
|    FB|2013| null|       6.009199444E7|
|    FB|2013|    1|        7.98024619E7|
|    FB|2013|    2|       5.040209474E7|
|    FB|2013|    3|         3.6359025E7|
|    FB|2013|    4|          3.35686E7|
|    FB|2013|    5|       4.464067273E7|
|    FB|2013|    6|         3.9416575E7|
|    FB|2013|    7|       6.536441364E7|
|    FB|2013|    8|       6.113609545E7|
|    FB|2013|    9|          7.915419E7|
+------+----+-----+--------------------+
```

Cube

```
df = spark.read.options(header = True, inferSchema = True).csv("stocks")  \
    .withColumn("year", functions.expr("year(date)"))  \
    .withColumn("month", functions.expr("month(date)"))

df.cube("year", "month").agg(round(avg("volume"), 2)).show()
```

```
+----+-----+---------------------+
|year|month|round(avg(volume), 2)|
+----+-----+---------------------+
|2019|    5|           4227726.56|
|2001|    8|           3077287.66|
|null|   10|           5555386.53|
|2009|    6|           8502630.54|
|2013|    1|           5856816.95|
|2003|    5|           4537885.11|
|2005|    7|           4035136.61|
|2009|    9|           7513157.14|
|2008|    8|           6262727.19|
|2019|    2|           4797267.23|
|2001|    2|           3960334.41|
|2003|    4|           4373478.95|
|2005|    3|           4310410.43|
|2006|    8|           4354680.79|
|2006|    5|           5038476.24|
|2001| null|           4080235.33|
|null|    2|           5577798.62|
|2007|    7|           6299974.29|
|2016|   12|            4338344.6|
|2017|    5|           4137718.26|
+----+-----+---------------------+
```

Window functions

	SQL	DataFrame API
Ranking functions	rank	rank
	dense_rank	denseRank
	percent_rank	percentRank
	ntile	ntile
	row_number	rowNumber
Analytic functions	cume_dist	cumeDist
	first_value	firstValue
	last_value	lastValue
	lag	lag
	lead	lead

https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html

Window function - lag

```
from pyspark.sql.window import Window

windowSpec = (Window
.partitionBy(df['symbol']).orderBy(df['date'])
.rowsBetween(-3, -1))

(df.selectExpr("*", "year(date) year", "month(date) month")
.withColumn("rolling_mean", F.mean("adjclose").over(windowSpec))
.orderBy("date")
.select("date", "symbol", "adjclose", "rolling_mean")
.where("symbol like 'FB'")
)
```

Window functions - rolling mean of period 3

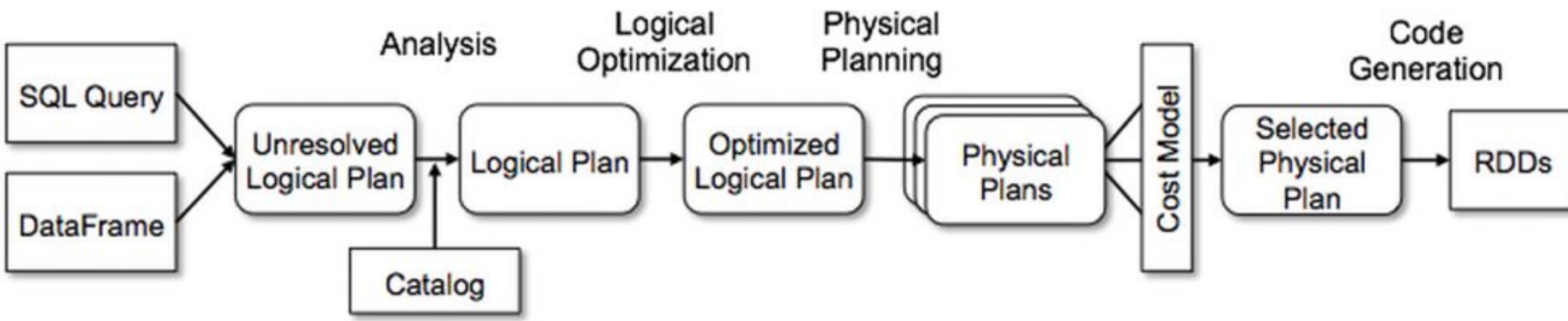
```
period = 3
windowSpec = Window.partitionBy(df['symbol']).orderBy(df['date']).rowsBetween(-period, -1)

df.selectExpr("symbol","date", "round(adjclose,2) adjclose") \
.withColumn("rolling_avg", round(avg("adjclose").over(windowSpec), 3)) \
.show(10)
```

```
+------+-------------------+--------+-----------+
|symbol|               date|adjclose|rolling_avg|
+------+-------------------+--------+-----------+
|  ALXN|2000-07-17 00:00:00|   19.94|       null|
|  ALXN|2000-07-18 00:00:00|   19.47|      19.94|
|  ALXN|2000-07-19 00:00:00|    19.0|     19.705|
|  ALXN|2000-07-20 00:00:00|   19.47|      19.47|
|  ALXN|2000-07-21 00:00:00|    19.5|     19.313|
|  ALXN|2000-07-24 00:00:00|   19.91|     19.323|
|  ALXN|2000-07-25 00:00:00|   18.28|     19.627|
|  ALXN|2000-07-26 00:00:00|   17.53|      19.23|
|  ALXN|2000-07-27 00:00:00|   16.81|     18.573|
|  ALXN|2000-07-28 00:00:00|    15.5|      17.54|
+------+-------------------+--------+-----------+
```

Window - last n days

```
df.createOrReplaceTempView("stock")
spark.sql("""
SELECT symbol,
       date,
       adjclose,
       AVG(adjclose) OVER (PARTITION BY symbol ORDER BY date
            RANGE BETWEEN INTERVAL 7 DAYS PRECEDING AND CURRENT ROW) AS rolling_avg_7_days
FROM stock
""").show()
```

Execution Plan



Quiz

1. Dataframe allows DML operations like insert, update, delete?
   a. True
   b. False
2. DataFrame allows
   a. DataFrame DSL operations
   b. SQL operations
   c. RDD operations
3. DataFrame DSL and SQL operation are equal in terms of execution performance
   a. True
   b. False
4. Why does custom UDF in PySpark slow down the dataframe operation?
5. Which data formats can be loaded as dataframe using spark packages?
   a. CSV
   b. XML
   c. Plain text
   d. Json

Query Solr

Check compatible driver https://github.com/lucidworks/spark-solr

Download shaded jar https://repo1.maven.org/maven2/com/lucidworks/spark/spark-solr/3.10.0/

Start pyspark

```
$ pyspark --jars spark-solr-3.10.0-shaded.jar --verbose
```

```
>>> opts = dict(flatten_multivalued = "true", collection = "credx_emails", zkhost = 
"sa01:2181,sa02:2181,sa03:2181/solr",request_handler = "/select")
```

```
>>> df = spark.read.format("solr").options(**opts).load()
```

Spark as distributed query engine

Set port and binding address

```
export HIVE_SERVER2_THRIFT_PORT=10000
export HIVE_SERVER2_THRIFT_BIND_HOST=0.0.0.0
```

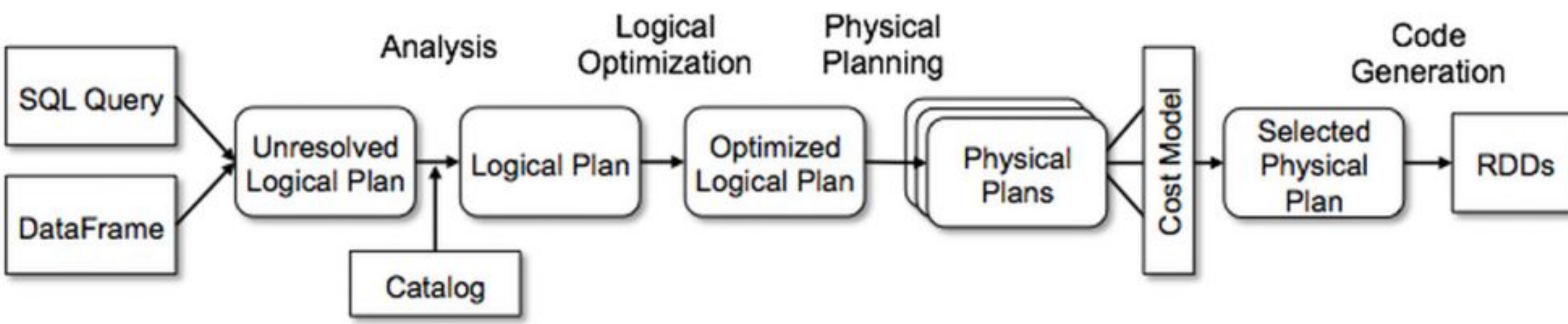
Start thrift service

```
$SPARK_HOME/sbin/start-thriftserver.sh
$SPARK_HOME/bin/beeline -u jdbc:hive2://demo1:10000/
```

Catalyst

Catalyst is based on functional programming constructs in Scala

- Easily add new optimization techniques and features to Spark SQL
- Enable external developers to extend the optimizer (e.g. adding data source specific rules, support for new data types, etc.)



http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf
https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

Catalysts Optimizer

- Analysis
- Logical Optimization
- Physical Planning
- Cost Model Analysis
- Code Generation

- Unresolved logical plan
- Resolved logical plan
- Optimized logical plan
- Physical plans

The goal of all these operations and plans is to produce automatically the most effective way to process your query.

Tungsten

Aim of Tungsten project is to substantially improve the efficiency of memory and CPU for Spark applications, to push performance closer to the limits of modern hardware.

- Memory Management and Binary Processing
- Cache-aware computation
- Whole stage code generation
- No virtual function dispatches
- Intermediate data in memory vs CPU registers
- Loop unrolling and SIMD

https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
https://spoddutur.github.io/spark-notes/second_generation_tungsten_engine.html

Out of memory issue (OOM)

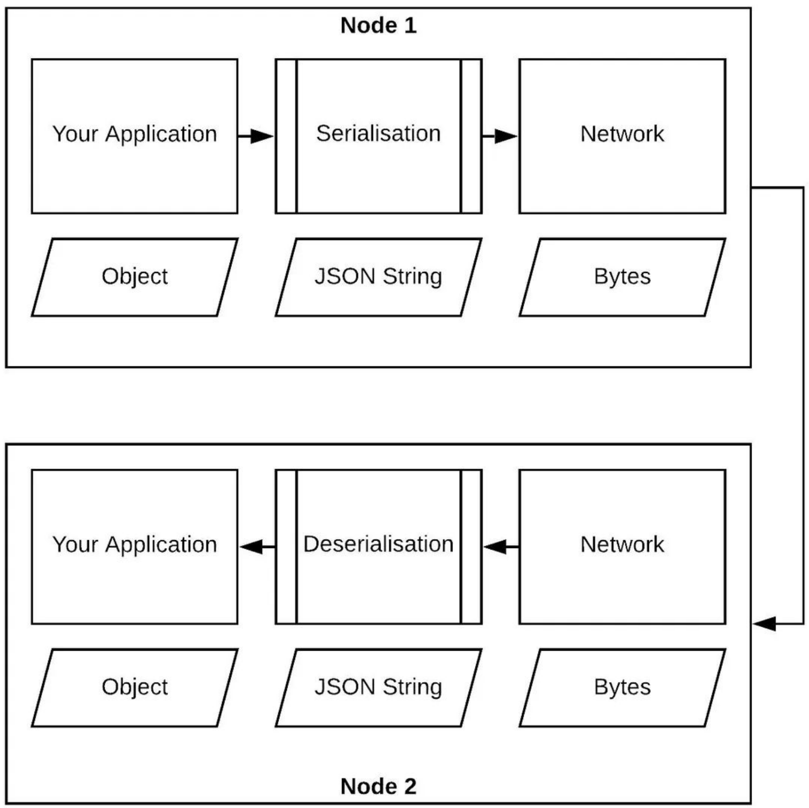
Driver memory issue

- collect
- broadcast large data

Executor memory issue

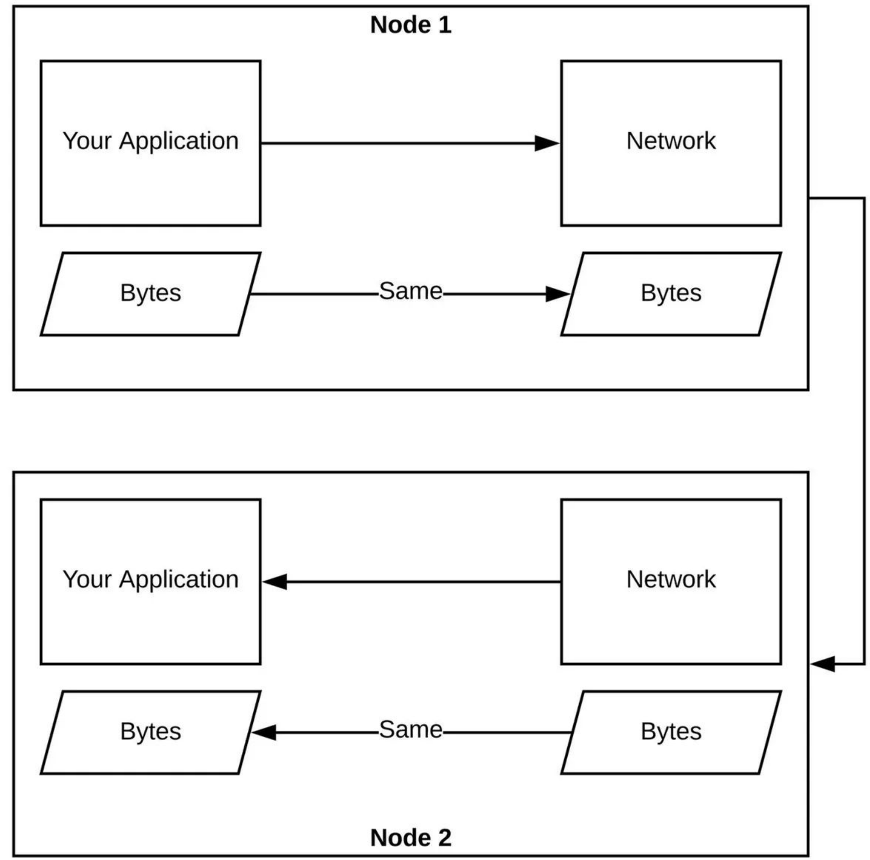
- big partition
- serialization (during cached or writing to parquet format)
- YARN memory overhead (this memory used for Python and R processes)
- large number of executor-cores per executor
- aggressive coalesce (leads to large partition)

Data sharing among processes



Zero copy data transfer

Zero-copy refers here to the fact that the bytes you application works on can be transferred over the wire without any modification. **All operations are carried out on byte buffers**.



Apache Arrow

Apache Arrow defines a language-independent **columnar memory format** for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs. The Arrow memory format also supports zero-copy reads for lightning-fast data access without serialization overhead.

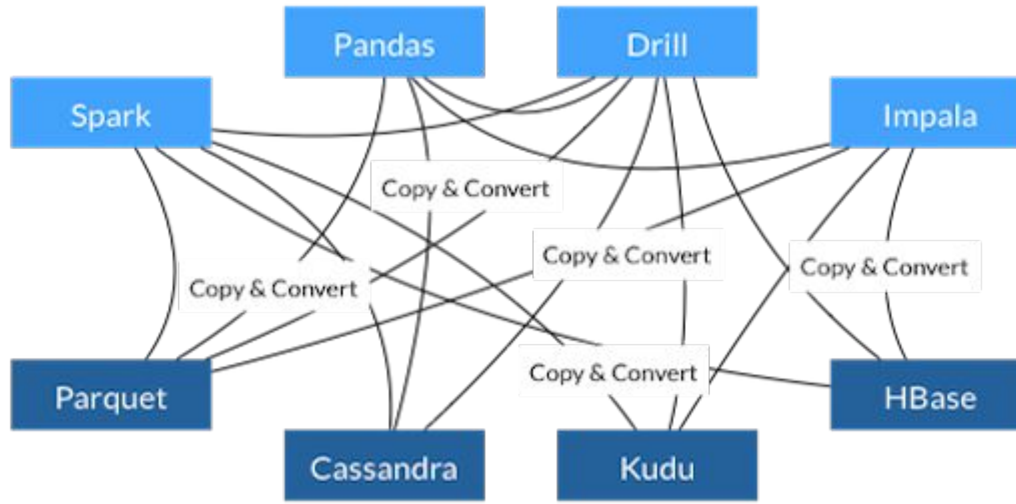
Languages: C, C++, Python, Java, Rust, etc.

Use cases:

- Reading/writing columnar storage formats (feather, parquet)
- Sharing memory locally across languages and processes
- In-memory data structure for analytics

Without Arrow

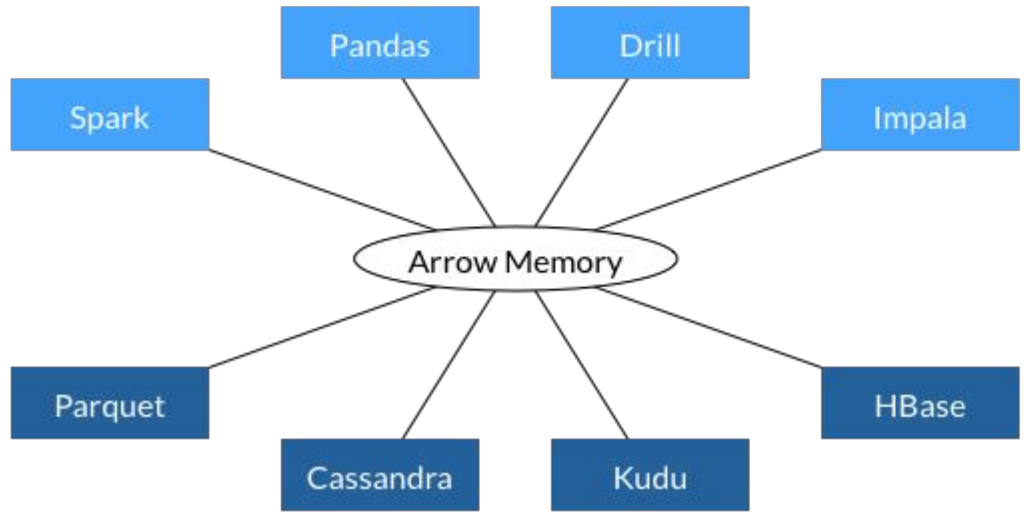
Without a standard columnar data format, every database and language has to implement its own internal data format.



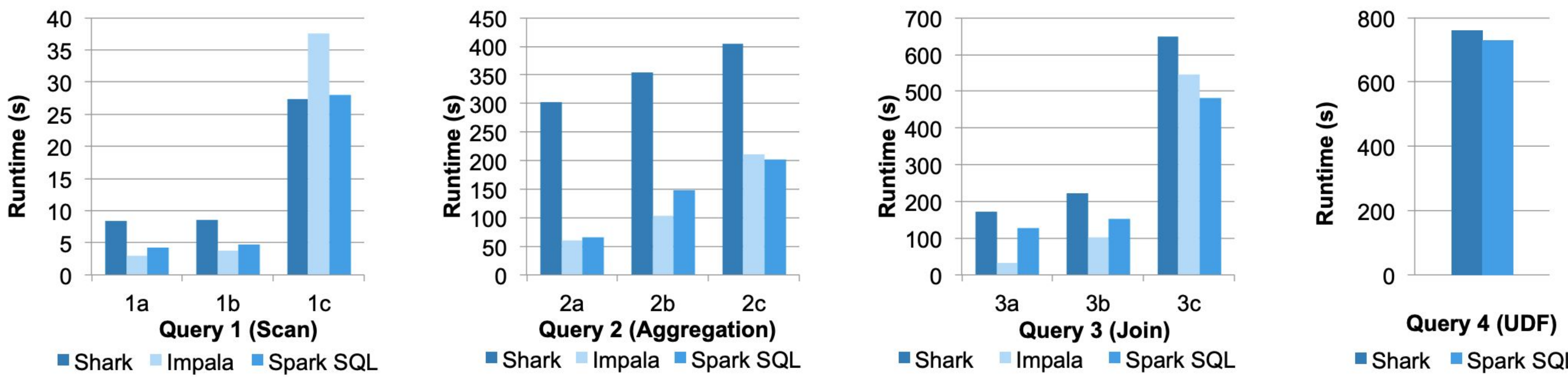
In some 90% of time is spent on data serialization/deserialization between processes.

With Arrow

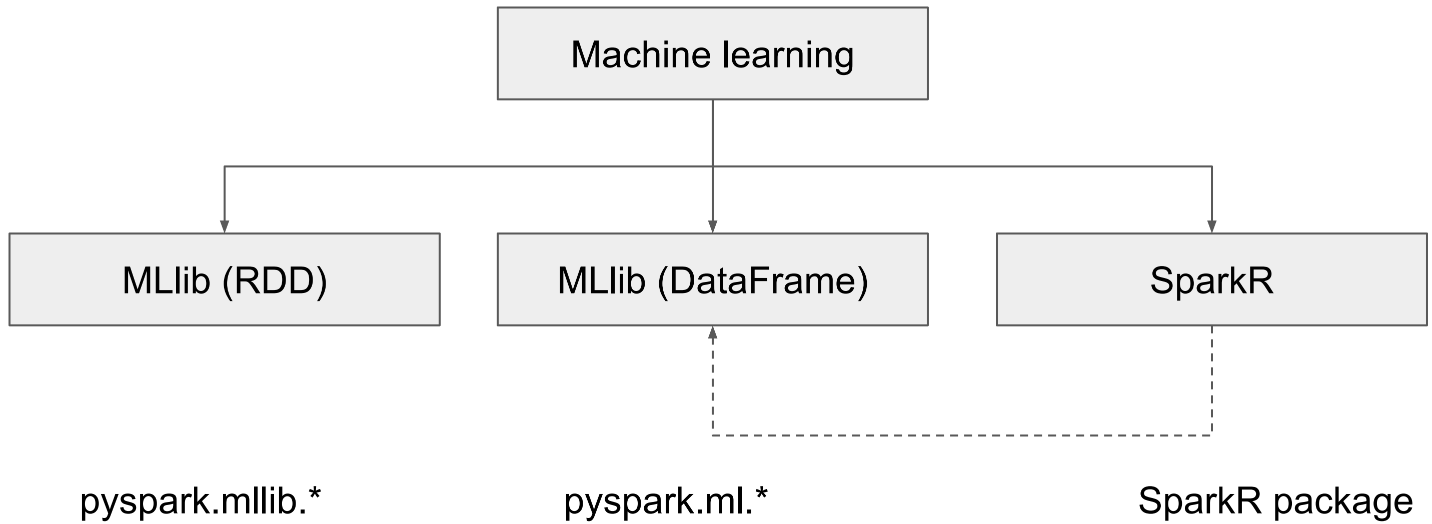
Arrow's in-memory columnar data format is an out-of-the-box solution to these problems. Systems that use or support Arrow can transfer data between them at little-to-no cost.



Performance comparison - spark vs shark and impala



Machine learning



Class Path Contains Multiple SLF4J Bindings

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_${scalaVersion}</artifactId>
    <version>${sparkVersion}</version>
    <scope>compile</scope>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
        <exclusion>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Spark internal article

https://www.kdnuggets.com/2020/07/monitoring-apache-spark-better-ui.html

Maximum number of files in HDFS

The maximum number of files in HDFS depends on the amount of memory available for the NameNode.

Each file object and each block object takes about 150 bytes of the memory. For example, if you have 10 million files and each file has 1 one block each, then you would need about 3GB of memory for the NameNode.

If you have 10 million files, each using a block, then we would be using: 10 million + 10 million = 20 million * 150 = 3,000,000,000 bytes = 3 GB MEMORY. Keep in mind the NameNode will need memory for other processes. So to support 10 million files then your NameNode will need much more than 3GB of memory.

HDFS Limits

`dfs.namenode.fs-limits.max-blocks-per-file`
`dfs.namenode.fs-limits.min-block-size`
`dfs.blocksize`

Improve join using bucket

https://towardsdatascience.com/best-practices-for-bucketing-in-spark-sql-ea9f23f7dd53

Adaptive Query Execution (AQE)

- Spark operators are often pipelined and executed in parallel processes. However, a shuffle or broadcast exchange breaks this pipeline.
- Adaptive Query Execution is available in Spark 3.0 or newer, and is enabled by default for Spark 3.2.0 and newer versions.
- AQE uses “statistics to choose the more efficient query execution plan” during the execution stages
- It attempts to re-optimization execution-specific rules
    - Dynamically coalescing shuffle partitions
    - Dynamically switching join strategies
    - Dynamically optimizing skew joins

https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html

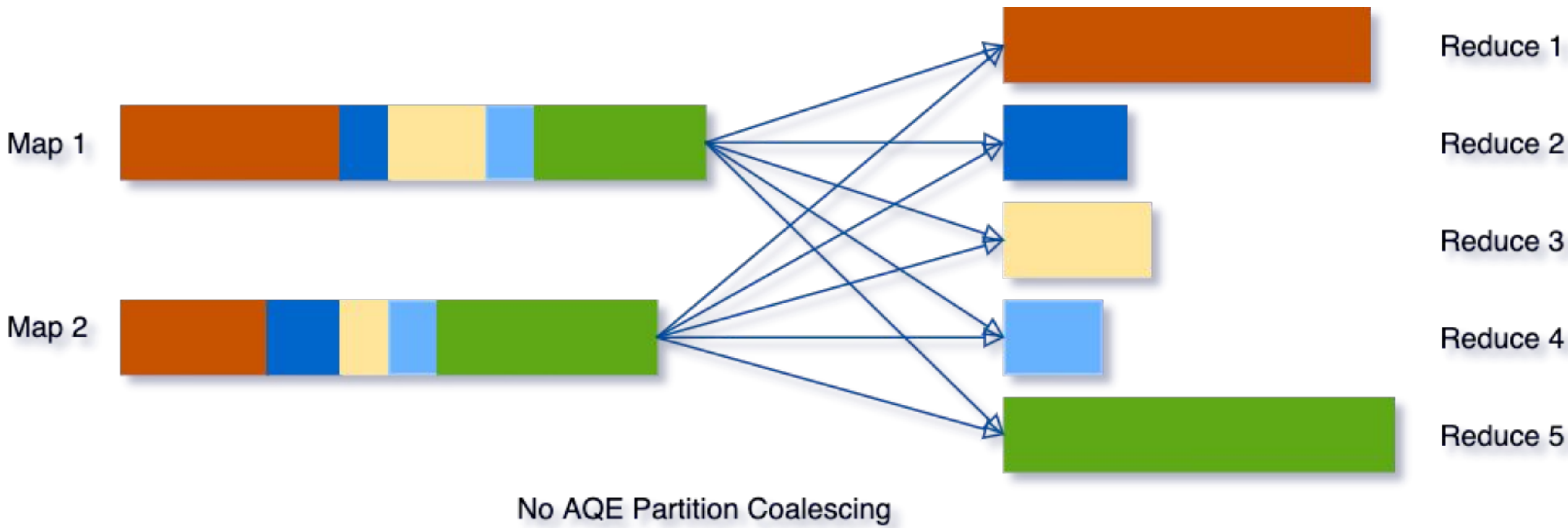
Why AQE is a game changer

- Before AQE - tools like Hive depends on table and column statistics to create optimized execution plan
- By making query optimization less dependent on static statistics, AQE has solved one of the greatest struggles of Spark cost-based optimization — the balance between the stats collection overhead and the estimation accuracy.
- AQE has largely eliminated the need for such statistics as well as for the manual tuning effort.
- AQE has also made SQL query optimization more resilient to the presence of arbitrary UDFs and unpredictable data set changes, e.g., sudden increase or decrease in data size, frequent and random data skew, etc. There's no need to "know" your data in advance any more. AQE will figure out the data and improve the query plan as the query runs, increasing query performance for faster analytics and system performance.

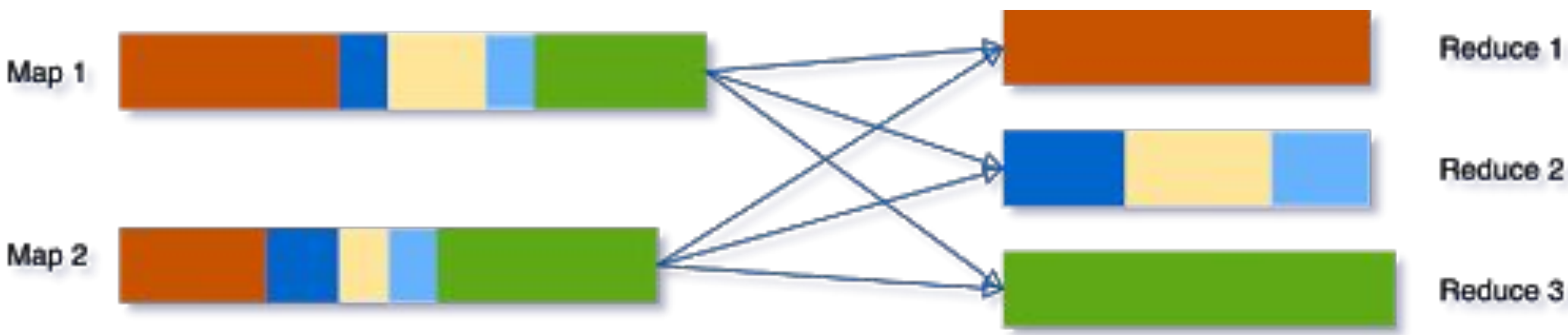
Dynamically coalescing shuffle partitions

- The best number of partitions is dependent on data and query
- If there are too few partitions, then the data size of each partition may be very large, and the tasks to process these large partitions may need to spill data to disk (e.g., when sort or aggregate is involved) and, as a result, slow down the query.
- If there are too many partitions, then the data size of each partition may be very small, and there will be a lot of small network data fetches to read the shuffle blocks, which can also slow down the query because of the inefficient I/O pattern. Having a large number of tasks also puts more burden on the Spark task scheduler.
- To solve this problem, we can set a relatively large number of shuffle partitions at the beginning, then combine adjacent small partitions into bigger partitions at runtime by looking at the shuffle file statistics

Hash partitioner shuffle



AQE Partitioning

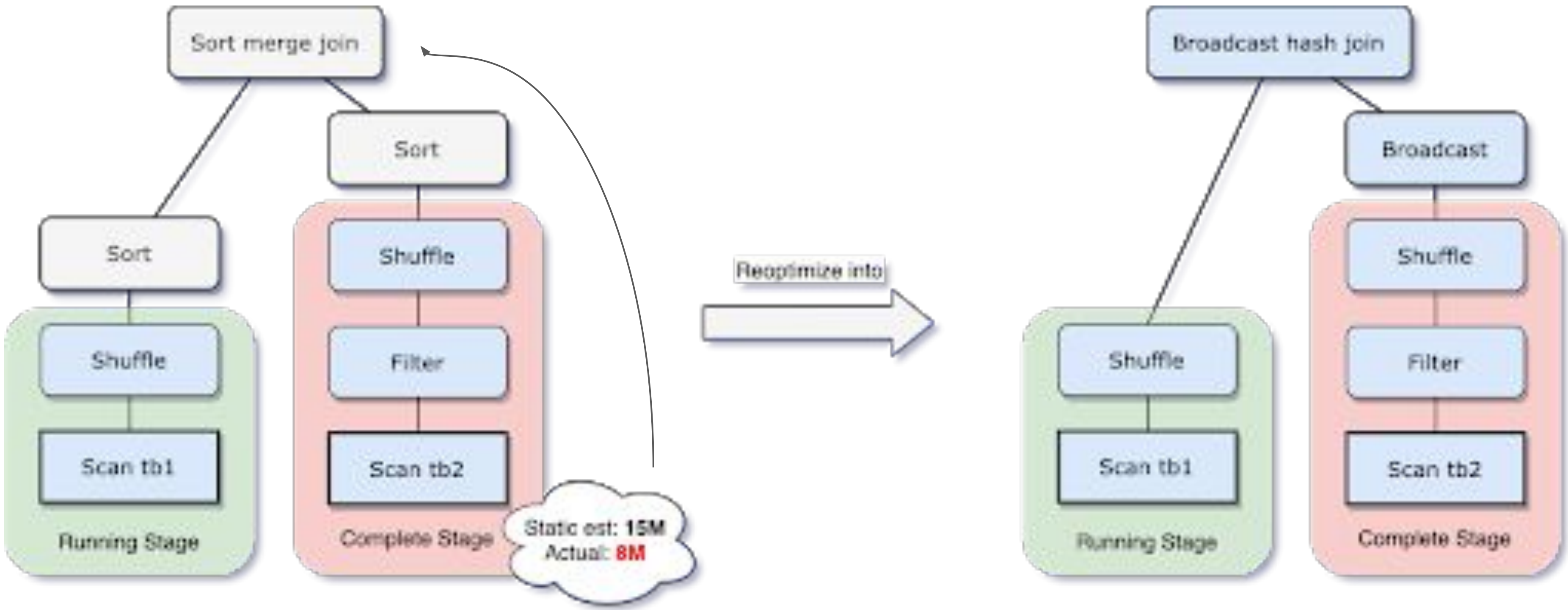


With AQE Partition Coalescing

Dynamically switching join strategies

- Spark supports a number of join strategies, among which broadcast hash join is usually the most performant if one side of the join can fit well in memory.
- Spark plans a broadcast hash join if the estimated size of a join relation is lower than the broadcast-size threshold.
- The estimated can be affected by the presence of a very selective filter, the join relation being a series of complex operators other than just a scan.
- To solve this problem, AQE now replans the join strategy at runtime based on the most accurate join relation size.

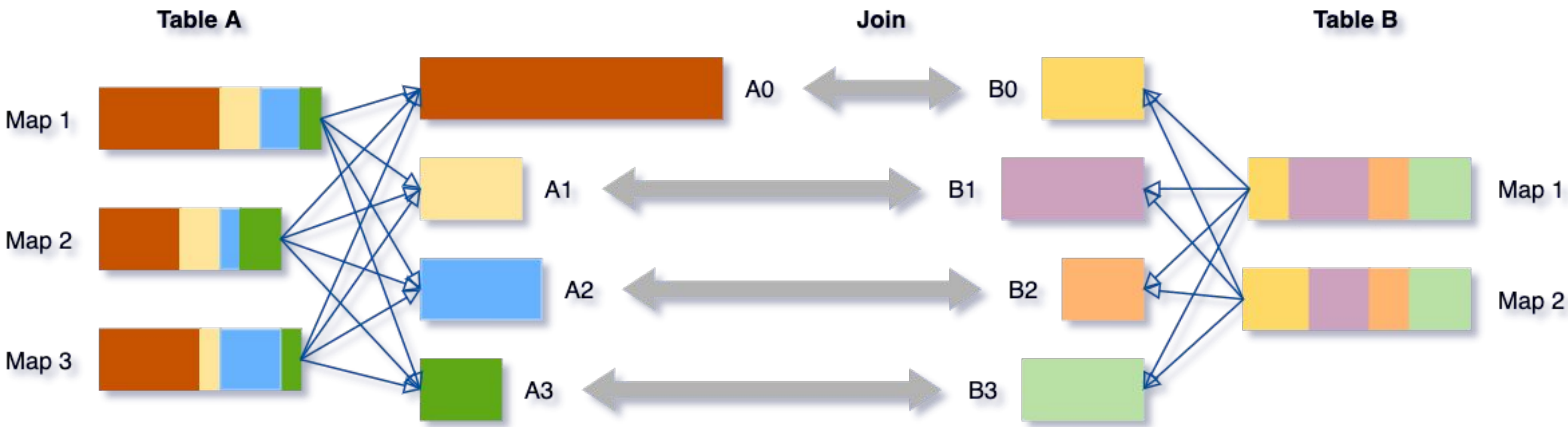
AQE: Adaptive join type based on the run time statistics



Dynamically optimizing skew joins

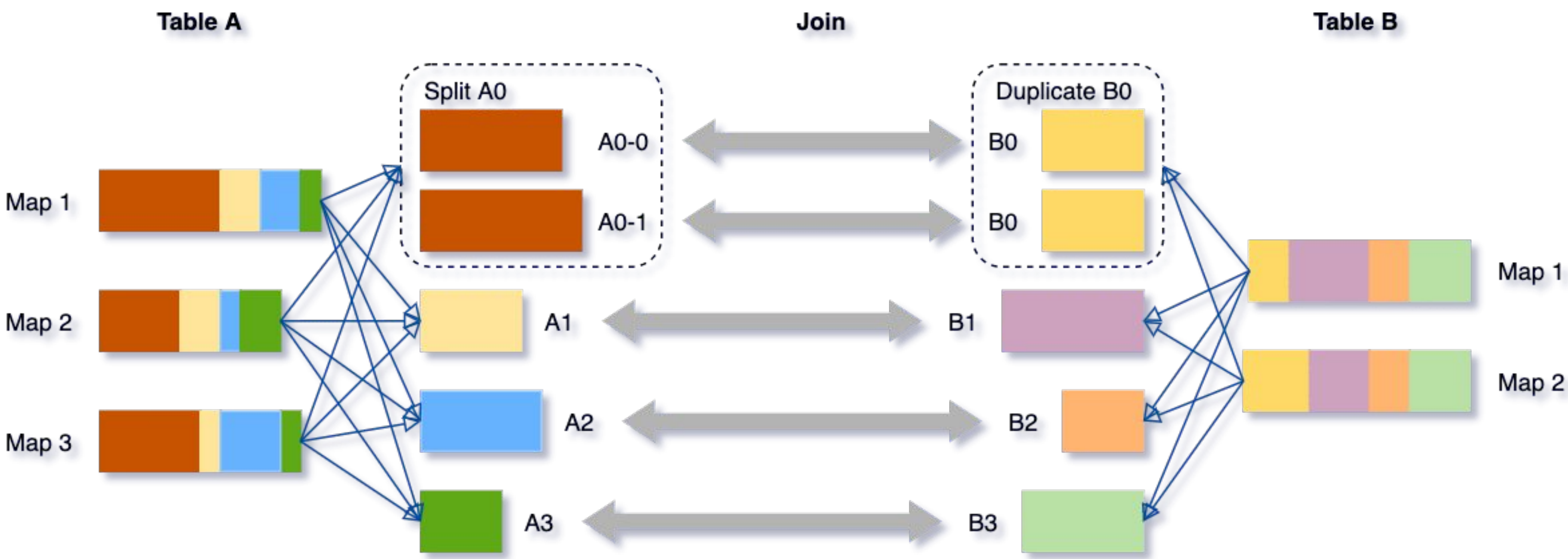
- Data skew occurs when data is unevenly distributed among partitions in the cluster.
- Severe skew can significantly downgrade query performance, especially with joins.
- AQE skew join optimization detects such skew automatically from shuffle file statistics. It then splits the skewed partitions into smaller subpartitions, which will be joined to the corresponding partition from the other side respectively.

No AQE Join



No AQE Skew Join

With AQE Join



With AQE Skew Join

Without this optimization, there would be four tasks running the sort merge join with one task taking a much longer time. After this optimization, there will be five tasks running the join, but each task will take roughly the same amount of time, resulting in an overall better performance.

AQE applicable if

- It is not a streaming query
- It contains at least one exchange (usually when there's a join, aggregate or window operator) or one subquery

AQE configuration

`spark.sql.adaptive.enabled` (default: true): Enable AQE

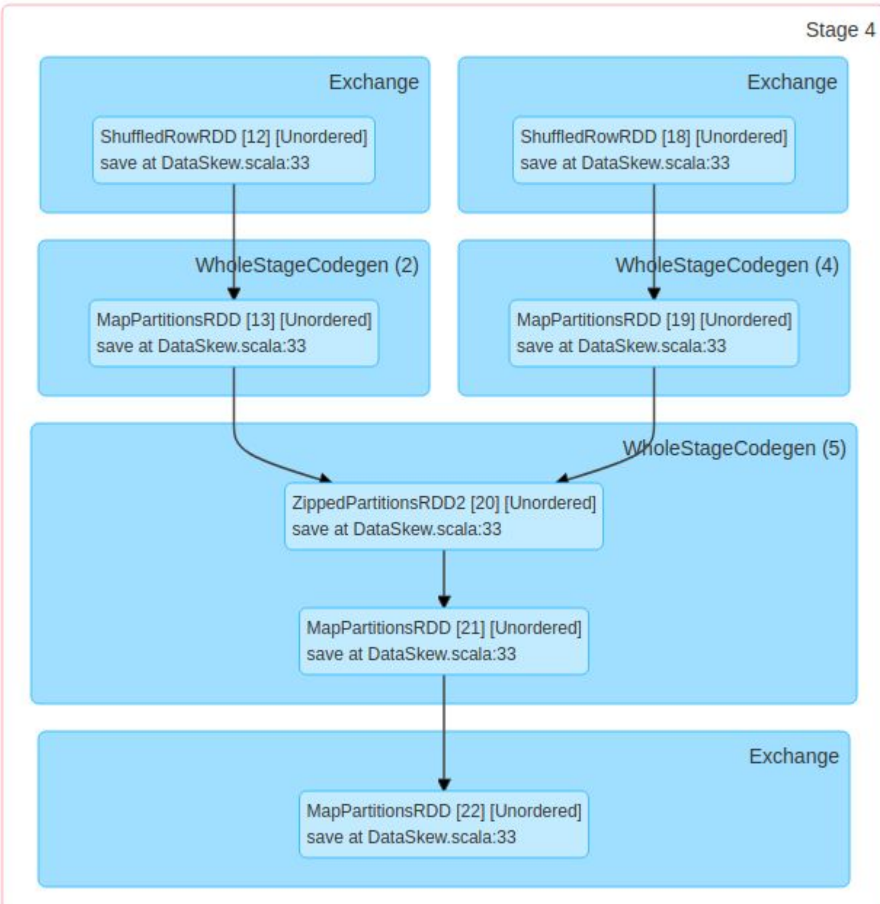
`spark.sql.adaptive.skewJoin.enabled` to True: Enable Skew Join Optimization

`spark.sql.adaptive.skewJoin.skewedPartitionFactor` (default value: 5). This adjusts the factor by which if medium partition size is multiplied, partitions are considered as skewed partitions if they are larger than that.

`spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes` (default value 256MB). This is the minimum size of skewed partition, and it marks partitions as skewed if they larger than the value set for this parameter and also are marked as skewed by the previous `spark.sql.adaptive.skewJoin.skewedPartitionFactor` param.

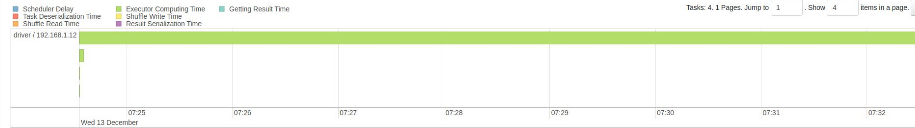
https://spark.apache.org/docs/latest/sql-performance-tuning.html#adaptive-query-execution

AQE - Disabled



▸ Show Additional Metrics
▾ Event Timeline
☐ Enable zooming

Tasks: 4. 1 Pages. Jump to 1 . Show 4 items in a page. Go



Summary Metrics for 4 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.2 s	0.2 s	2 s	8.1 min	8.1 min
GC Time	0.0 ms	0.0 ms	0.1 s	2 s	2 s
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	503.1 KiB / 50865	10 MiB / 959135	10 MiB / 959135
Shuffle Write Size / Records	0.0 B / 0	0.0 B / 0	202 B / 3	204 B / 3	204 B / 3

▸ **Aggregated Metrics by Executor**

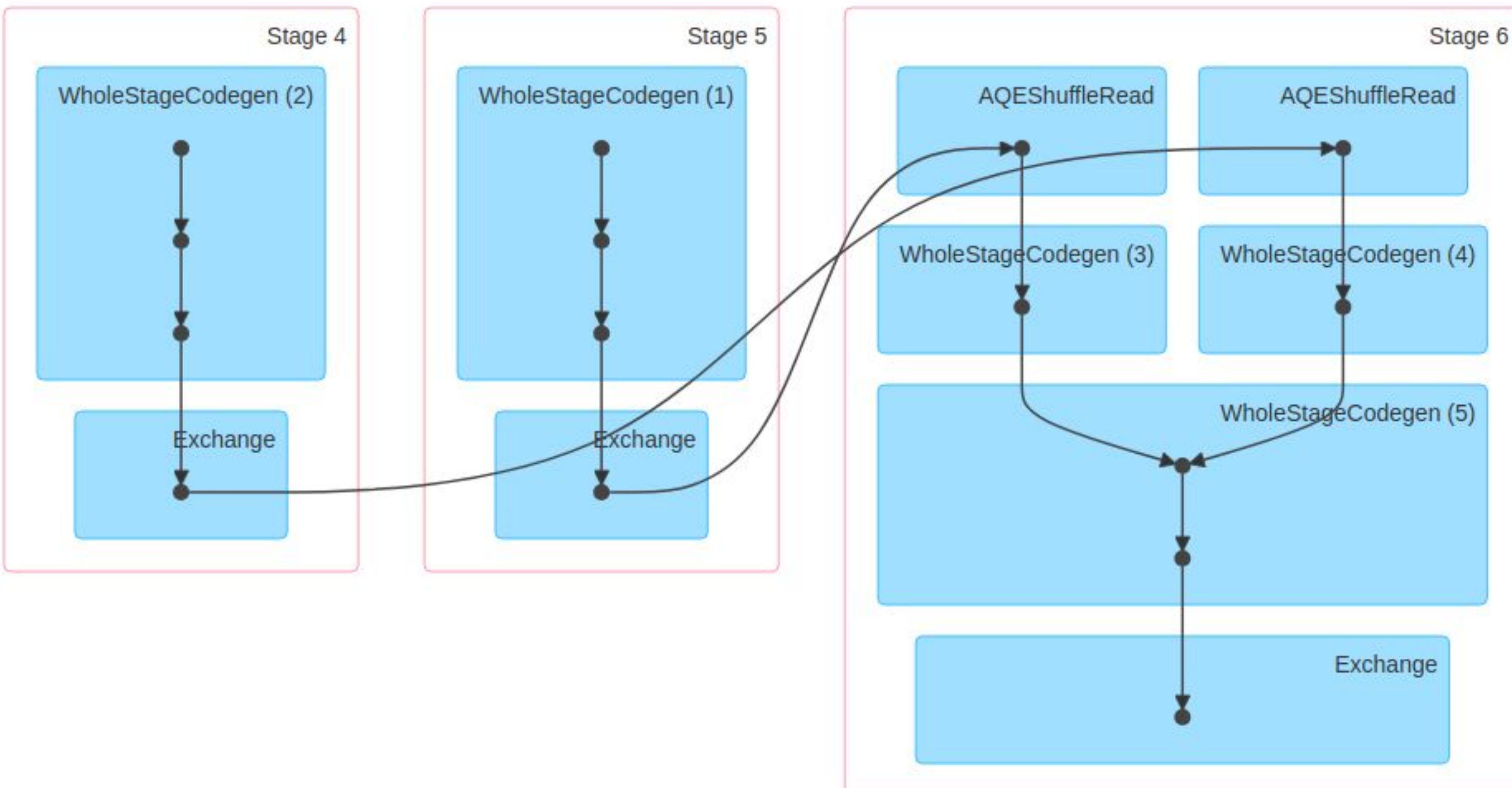
Tasks (4)

Show 20 entries   Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Write Size / Records	Shuffle Read Size / Records	Errors
0	24	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.12		2023-12-13 07:24:33	0.2 s				
1	22	0	SUCCESS	NODE_LOCAL	driver	192.168.1.12		2023-12-13 07:24:33	8.1 min	2 s	204 B / 3	10 MiB / 959135	
2	25	0	SUCCESS	PROCESS_LOCAL	driver	192.168.1.12		2023-12-13 07:24:33	0.2 s				
3	23	0	SUCCESS	NODE_LOCAL	driver	192.168.1.12		2023-12-13 07:24:33	2 s	0.1 s	202 B / 3	503.1 KiB / 50865	

Showing 1 to 4 of 4 entries   Previous 1 Next

AQE - Enabled



Stage 4
WholeStageCodegen (2)
Exchange
Stage 5
WholeStageCodegen (1)
Exchange
Stage 6
AQEShuffleRead
AQEShuffleRead
WholeStageCodegen (3)
WholeStageCodegen (4)
WholeStageCodegen (5)
Exchange

What is Koalas

The Koalas project makes data scientists more productive when interacting with big data, by implementing the pandas DataFrame API on top of Apache Spark.

Pandas is the de facto standard (single-node) DataFrame implementation in Python, while Spark is the de facto standard for big data processing. With this package, you can:

- Be immediately productive with Spark, with no learning curve, if you are already familiar with pandas.
- Have a single codebase that works both with pandas (tests, smaller datasets) and with Spark (distributed datasets).

Koalas as replacement for pandas?

With Koalas, we can mostly replace the existing pandas code with Koalas. ~90% features of Pandas dataframe are available in Koalas.

Lazy Vs Eager evaluation: Pandas are inherently eagerly evaluated but Koalas would use lazy evaluation ie all of the computations are done only when some actions such as count() or collect() are called. You might wanna keep that in mind when working with Koalas.

Like pandas, Koalas has two types of data structure - Series, and DataFrame

Koalas Dataframe

Koalas DataFrame

- Follow the structure of pandas

- Provide pandas APIs

- Implement index/identifier (index values may not necessarily be unique)

- Translates pandas APIs into logical plan of Spark SQL

- The execution plan is optimized by Spark SQL engine

Read files recursively

```
# conf.set("spark.hive.mapred.supports.subdirectories","true")


sc.hadoopConfiguration.set("mapreduce.input.fileinputformat.input.dir.recursive","true")
sc.textFile("path/*/*")
```

Koalas Index Type

	Distributed computation	Map-side join	Continuous increment	Performance
sequence	No, in a single worker node	No, requires a shuffle	Yes	Bad for large dataset
distributed_sequence	Yes	Yes, but requires another job	Yes	Good Enough
distributed *	Yes	Yes	No	Good

```
ks.set_option("compute.default_index_type", "sequence")
ks.reset_option("compute.default_index_type")
ks.get_option("compute.default_index_type")
```

`sequence` should not be used for large dataset
`distributed/distributed_sequence` is suitable for most of the cases

https://koalas.readthedocs.io/en/latest/user_guide/options.html#default-index-type

Distributed sequence

- Spark will create an id column for a dataframe.
- Id is a long data types, hence 64 bit integer
- 31 bits is used for partition id and remaining 33 bits are used for generating a sequence for each record
- Limits: 31 bits allow ~ 2 billion partitions per dataframe, 8 billion records per partition

pyspark.sql.functions.monotonically_increasing_id

Thank You