Apache Spark

Saurav Agarwal

WHAT IS APACHE SPARK

What is Apache Spark

Is a fast in-memory data-processing engine

Ability to efficiently execute streaming, machine-learning or SQL workloads which require fast-iterative access to data-sets

Can run on top of Apache Hadoop YARN, Mesos & Kubernetes

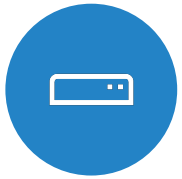
Is designed for data-science and its abstraction makes data-science easier

It can cache data-set in memory and speed up iterative data-processing

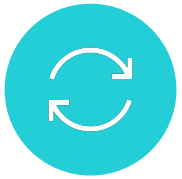
Includes ML-lib

Is 100 folds faster than MR, in benchmark tests

Hadoop Mapreduce Limitations



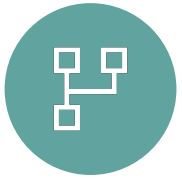
It's based on disk based computing



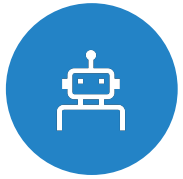
Suitable for single pass computations -not iterative computations

Needs a sequence of MR jobs to run iterative tasks

Needs integration with several other frameworks/tools to solve big data use cases



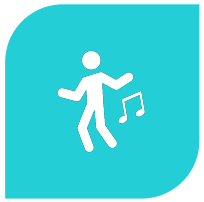
Need Apache Storm for stream data processing



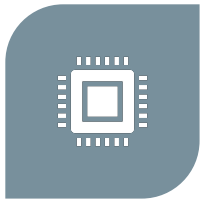
Need Apache Mahout for machine learning

Performance

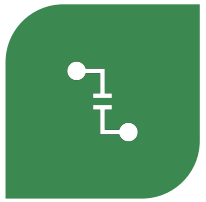
SPARK PROCESSES DATA IN MEMORY, WHILE MR PERSISTS BACK TO DISK, AFTER A MAPREDUCE JOB



SO SPARK SHOULD OUTPERFORM MR



NONETHELESS, SPARK NEEDS A LOT OF MEMORY



IF DATA IS TOO BIG TO FIT IN MEMORY, THEN THERE WILL BE MAJOR PERFORMANCE DEGRADATION FOR SPARK

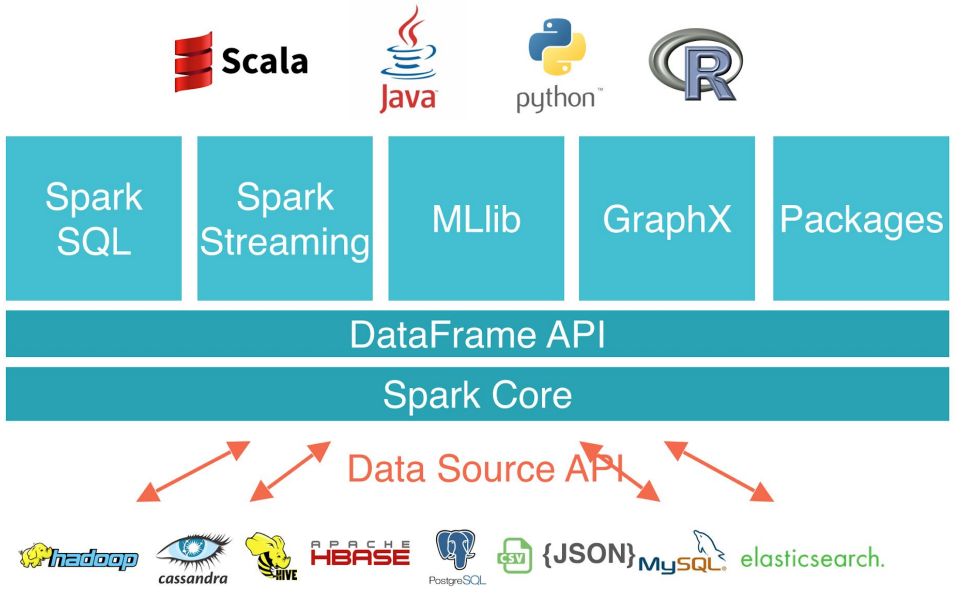
MR KILLS ITS JOB, AS SOON AS IT'S DONE

SO IT CAN RUN EASILY ALONGSIDE OTHER SERVICES WITH MINOR PERFORMANCE DIFFERENCES

STILL SPARK HAS AN ADVANTAGE, AS LONG AS WE ARE TALKING ABOUT ITERATIVE OPERATIONS ON THE DATA

Spark Components

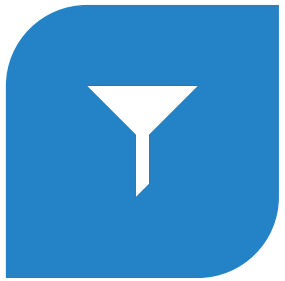
- **Spark Core,RDD, DF, DS, SQL –** API's for basic data processing needs for batch layer.
- **Spark Streaming-** API for real time processing needs for speed layer of data pipeline.
- **Spark MLLib –** API for Machine learning processing needs
- **Graph X-** API for needs of complex processing of Graph based data models with nodes and interactions between them.



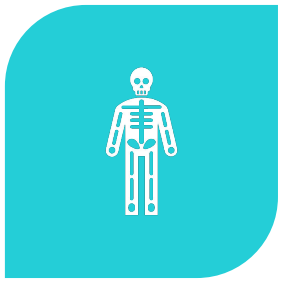
RESILIENT DISTRIBUTED DATASET (RDD)

Saurav Agarwal

What is a RDD?

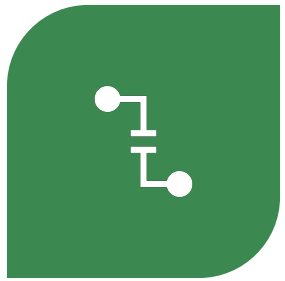


IT IS THE PRIMARY ABSTRACTION IN SPARK AND IS THE CORE OF APACHE SPARK



ONE COULD COMPARE RDDS TO COLLECTIONS IN PROGRAMMING, A RDD IS COMPUTED ON MANY JVMS WHILE A COLLECTION LIVES ON A SINGLE JVM

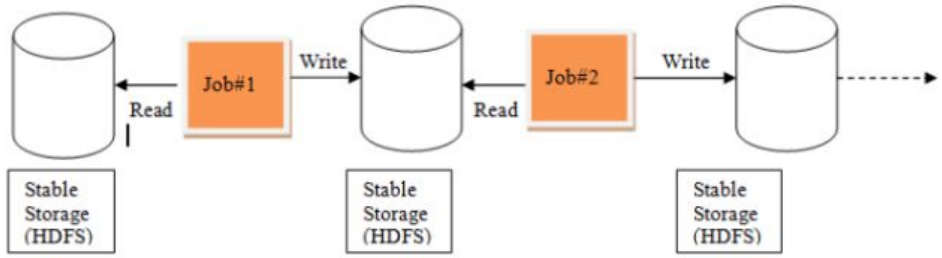
IMMUTABLE AND PARTITIONED COLLECTION OF RECORDS



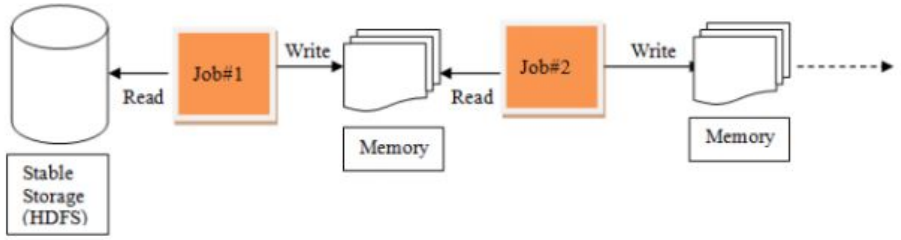
CAN ONLY BE CREATED BY READING DATA FROM A STABLE STORAGE LIKE HDFS OR BY TRANSFORMATIONS ON EXISTING RDD'S

Features of RDD

- For iterative distributed computing, it's common to reuse and share data among multiple jobs or do parallel ad-hoc queries over a shared dataset
- The persistent issue with data reuse or data sharing exists in distributed computing system(like MR) -that is, you need to store data in some intermediate stable distributed store such as HDFS or Amazon S3
- This makes overall computation of jobs slower as it involves multiple IO operations, replications and serializations in the process
- Resilient, fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures
- Distributed with data residing on multiple nodes in a cluster
- Dataset is a collection of partitioned data with primitive values or values of values, e.g tuples or other objects



Iterative processing in MR



Iterative processing in Spark

Advantages of RDD

- **How Are RDD's Fault Tolerant?**
- As RDD's are created over a set of transformations, it logs these transformations rather than actual data
- Graph of transformations to produce one RDD is called a Lineage Graph
- **For example** -firstRDD=spark.textFile("hdfs://... ")
- secondRDD=firstRDD.filter(someFunction)
- thirdRDD = secondRDD.map(someFunction)
- In case of we lose some partition of RDD , we can replay the transformation on that partition in lineage to achieve the same computation
- This is the biggest benefit of RDD , because it saves a lot of efforts in data management and replication and thus achieves faster computations

Traits of RDD

- In-Memory , data inside RDD is stored in memory as much (size) and long (time) as possible
- Immutable or Read-Only , it does not change once created and can only be transformed using transformations to new RDDs
- Lazy evaluated , the data inside RDD is not available or transformed until an action is executed that triggers the execution
- Cacheable , you can hold all the data in a persistent “storage” like memory (default and the most preferred) or disk (the least preferred due to access speed)
- Parallel, process data in parallel
- Typed-RDD records have types, Long in RDD[Long] or (Int, String) in RDD[(Int, String)]
- Partitioned-records are partitioned (split into logical partitions) and distributed across nodes in a cluster
- Location-Stickiness-RDD can define placement preferences to compute partitions (as close to the records as possible)
- RDD Supports Two Kinds of Operations
- **Actions** -operations that trigger computation and return values
- **Transformations** -lazy operations that return another RDD

Creating a RDD

Create RDDs from Python collections (lists)

```
>>>  data = [1,2,3,4,5]
>>> rDD=sc.parallelize(data,4)
>>>rDD
```

No computation occurs with sc.parallelize()
Spark only records how to create the RDD with four partitions

From HDFS, text files, Hypertable, Amazon S3, Apache Hbase,SequenceFiles, any other Hadoop InputFormat, and directory or glob wildcard: /data/201404*

```
>>> distFile = sc.textFile("README.md", 4)
>>> distFile
```

RDD Actions

Saurav Agarwal

ACTIONS on RDD

Actions are RDD operations that produce non-RDD values

In other words, a RDD operation that returns a value of any type except RDD[T] is an action

They trigger execution of RDD transformations to return values

Simply put, an action evaluates the RDD lineage graph

You can think of actions as a valve and until action is fired, the data to be processed is not even in the pipes, transformations

Only actions can materialize the entire processing pipeline with real data Actions are one of two ways to send data from executors to the driver (the other being accumulators)

Actions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

Getting Data Out of RDDs

```
>>> rdd=sc.parallelize([1,2,3])
>>> rdd.reduce(lambda a,b:a*b)

>>> rdd.take(2)
>>> rdd.collect()

>>>rdd =sc.parallelize([5,3,1,2])
>>>rdd.takeOrdered(3,lambda s: -1 * s)
```

```
lines=sc.textFile("...",4)
print lines.count()
```

count() causes Spark to: read data , sum within partitions, combine sums in driver

saveAsTextFile(path) - Save this RDD as a text file, using string representations of elements.

RDD Transformations

Saurav Agarwal

Transformations of RDD

Transformations are lazy operations on a RDD that create one or many new RDDs, e.g. map, filter, reduceByKey, join, cogroup, randomSplit

They are functions that take a RDD as the input and produce one or many RDDs as the output They do not change the input RDD (since RDDs are immutable), but always produce one or more new RDDs by applying the computations they represent

Transformations are lazy,they are not executed immediately but only after calling an action are transformations executed

After executing a transformation, the result RDD(s) will always be different from their parents and can be smaller (e.g. filter, count, distinct, sample), bigger (e.g flatMap, union, cartesian) or the same size (e.g. map)

RDD Spark Transformations

Transformation	Description
`map(`*`func`*`)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(`*`func`*`)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([`*`numTasks`*`]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(`*`func`*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

Transformations

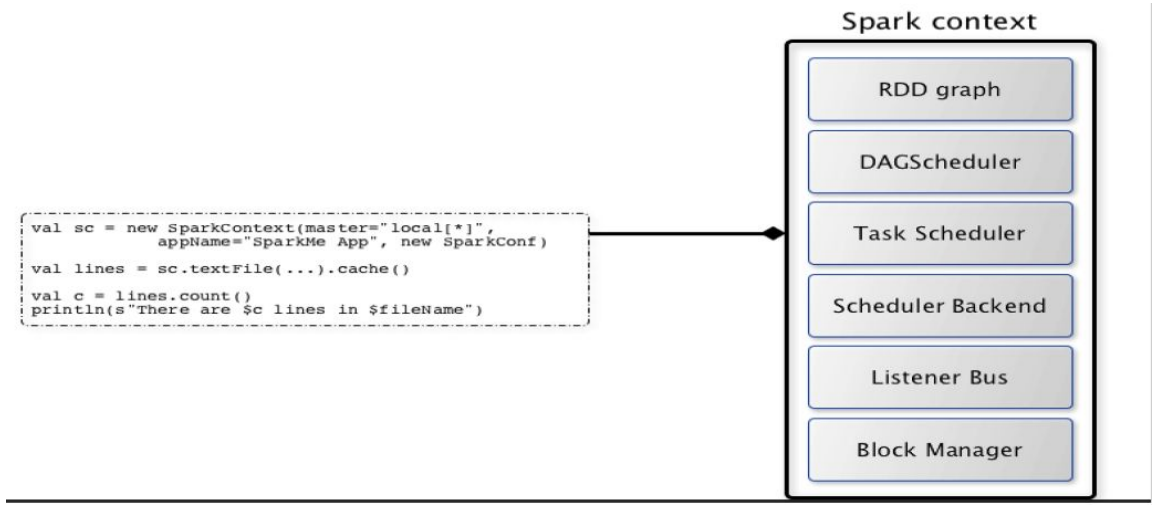
```
>>> rdd=sc.parallelize([1,2,3,4])
>>> rdd.map(lambda x:x*2)

>>> rdd.filter(lambda x:x%2==0)

>>> rdd2=sc.parallelize([1,4,2,2,3])
>>> rdd2.distinct()
```

SparkSession & SparkContext

Saurav Agarwal

SparkContext

- It is the entry point to spark core -heart of the spark application
- Spark context sets up internal services and establishes a connection to a Spark execution environment
- Once a SparkContext is created you can use it to create RDDs, accumulators and broadcast variables, access Spark services and run jobs (until SparkContext is stopped)
- A Spark context is essentially a client of Spark's execution environment and acts as the master of your Spark application (don't get confused with the other meaning of Master in Spark, though)



SparkContext & SparkSession

Prior to spark 2.0.0

- **sparkContext** was used as a channel to access all spark functionality.
- The spark driver program uses spark context to connect to the cluster through a resource manager (YARN or Mesos..).
- **sparkConf** is required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node
- In order to use APIs of **SQL,HIVE , and Streaming**, separate contexts need to be created.

like

```
val conf=newSparkConf()

val sc = new SparkContext(conf)

val hc = new hiveContext(sc)

val ssc = new streamingContext(sc).
```

SPARK 2.0.0 onwards

- **SparkSession** provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with **Dataframe** and **Dataset** APIs. All the functionality available with **sparkContext** are also available in **sparkSession**.
- In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.
- Once the SparkSession is instantiated, we can configure Spark's run-time config properties

CREATING SPARKCONTEXT INSTANCE

- You can create a SparkContext instance with or without creating a SparkConf object first

```
>>> spark = SparkSession.builder \
... .master("local") \
... .appName("Word Count") \
... .config("spark.some.config.option", "some-value") \
 ... .getOrCreate()
```

You can create a spark session using this:

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

OR

```
>>> from pyspark.conf import  SparkConf
>>> c = SparkConf()
>>> SparkSession.builder.config(conf=c)
```

From here
http://spark.apache.org/docs/2.0.0/api/python/pyspark.sql.html

Spark Caching

Saurav Agarwal

Caching

- It is one mechanism to speed up applications that access the same RDD multiple times.
- An RDD that is not cached, nor checkpointed, is re-evaluated again each time an action is invoked on that RDD.
- There are two function calls for caching an RDD: cache() and persist(level: StorageLevel).
- The difference among them is that cache() will cache the RDD into memory, whereas persist(level) can cache in memory, on disk, or off-heap memory according to the caching strategy specified by level.
- persist() without an argument is equivalent with cache(). Freeing up space from the Storage memory is performed by unpersist().

When to use caching

It is recommended to use caching in the following situations:

- RDD re-use in iterative machine learning applications
- RDD re-use in standalone Spark applications
- When RDD computation is expensive, caching can help in reducing the cost of recovery in the case one executor fails

Caching RDDs

```
lines=sc.textFile("...",4)
lines.cache()
#save,don't recompute!
comments=lines.filter(isComment)
print lines.count(),comments.count()
```

Storage Level defines how a RDD is persisted.

rdd.persist(StorageLevel. MEMORY_ONLY) is same as rdd.cache()

- NONE (default)
- DISK_ONLY
- MEMORY_ONLY (default for cache operation for RDD's)
- MEMORY_AND_DISK
- MEMORY_AND_DISK_SER

Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.

Spark Pair RDD

Saurav Agarwal

Spark Key-Value RDDs

Similar to Map Reduce, Spark supports Key-Value pairs

Each element of a Pair RDD is a pair tuple

```
>>>rdd=sc.parallelize([(1,2),(3,4)])
```

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Spark Key-Value RDDs

```
>>> rdd = sc.parallelize([(1,2),(3,4),(3,6)])
>>> rdd.reduceByKey(lambda a,b:a+b)
>>>rdd2=sc.parallelize([(1,'a'),(2,'c'),(1,'b')])
>>>rdd2.sortByKey()
>>> rdd2=sc.parallelize([(1,'a'),(2,'c'),(1,'b')])
>>> rdd2.groupByKey()
```

Be careful using groupByKey() as it can cause a lot of data movement across the network and create large Iterables at workers

Pair RDD Joins

X.join(Y)

» Return RDD of all pairs of elements with matching keys in X and Y

» Each pair is (k, (v1, v2)) tuple, where (k, v1) is in X and (k, v2) is in Y

```
>>> x=sc.parallelize([("a",1),("b",4)])
>>> y=sc.parallelize([("a",2),("a",3)])
>>> sorted(x.join(y).collect())
```

X.fullOuterJoin(Y)

» For each element (k, v) in X, resulting RDD will either contain

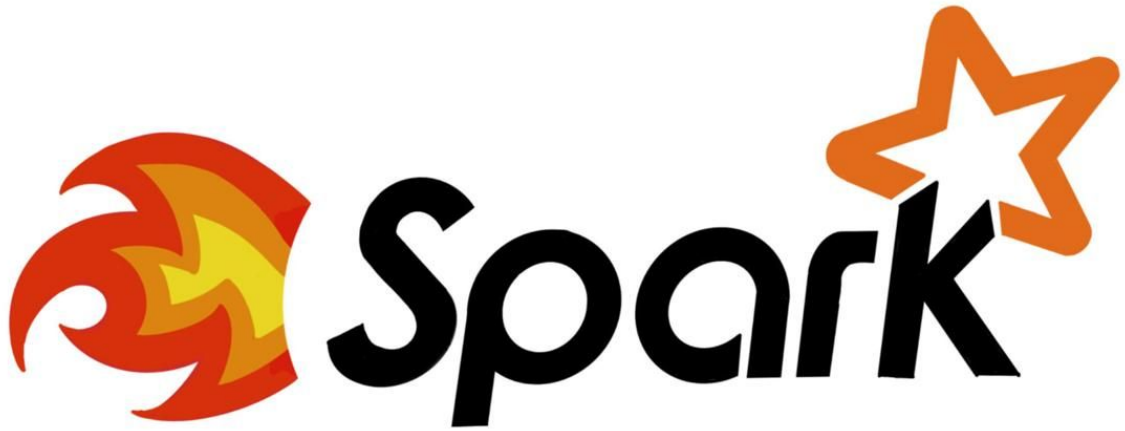
All pairs (k, (v, w)) for w in Y, or (k, (v, None)) if no elements in Y have k

» For each element (k, w) in Y, resulting RDD will either contain

All pairs (k, (v, w)) for v in X, or (k, (None, w)) if no elements in X have k

```
>>> x=sc.parallelize([("a",1),("b",4)])
>>> y=sc.parallelize([("a",2),("c",8)])
>>> sorted(x.fullOuterJoin(y).collect())
```

Spark Shared Variables



Saurav Agarwal

pySpark Shared Variables

Broadcast Variables

» Efficiently send large, read-only value to all workers

» Saved at workers for use in one or more Spark operations

» Like sending a large, read-only lookup table to all the nodes

At the driver >>>broadcastVar=sc.broadcast([1,2,3])

At a worker (in code passed via a closure) >>>broadcastVar.value

Accumulators

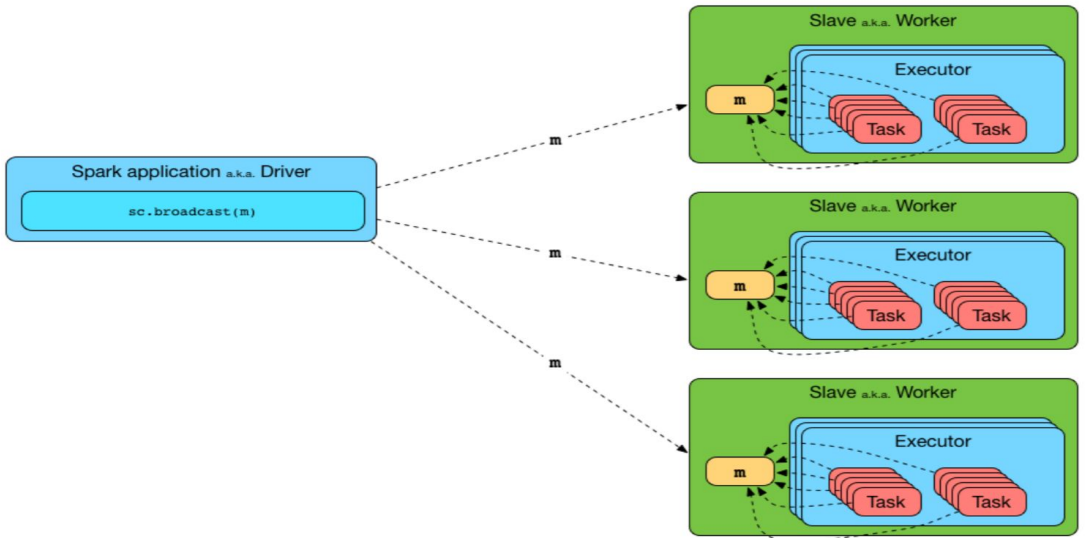
» Aggregate values from workers back to driver

» Only driver can access value of accumulator

» For tasks, accumulators are write-only

» Use to count errors seen in RDD across workers

```
>>>accum=sc.accumulator(0)
>>> rdd=sc.parallelize([1,2,3,4])
>>> def f(x):
>>>   global accum
>>>   accum+=x
>>>rdd.foreach(f)
>>>accum.value
```



Accumulators Example

Tasks at workers cannot access accumulator's values

Tasks see accumulators as write-only variables

Accumulators can be used in actions or transformations:

» Actions: each task's update to accumulator is applied only once

» Transformations: no guarantees (use only for debugging)

Types: integers, double, long, float

Counting Empty Lines

```
file=sc.textFile(inputFile)
#Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)
def extractCallSigns(line):
    global blankLines
    #Make the global variable accessible
    if(line==""):
      blankLines+=1
    return line.split("")
callSigns = file.flatMap(extractCallSigns)
print "Blank lines:%d" % blankLines.value
```

Spark Partitions, Coalesce and Repartition

Saurav Agarwal

PARTITIONS

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

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

COALESCE AND REPARTITION

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

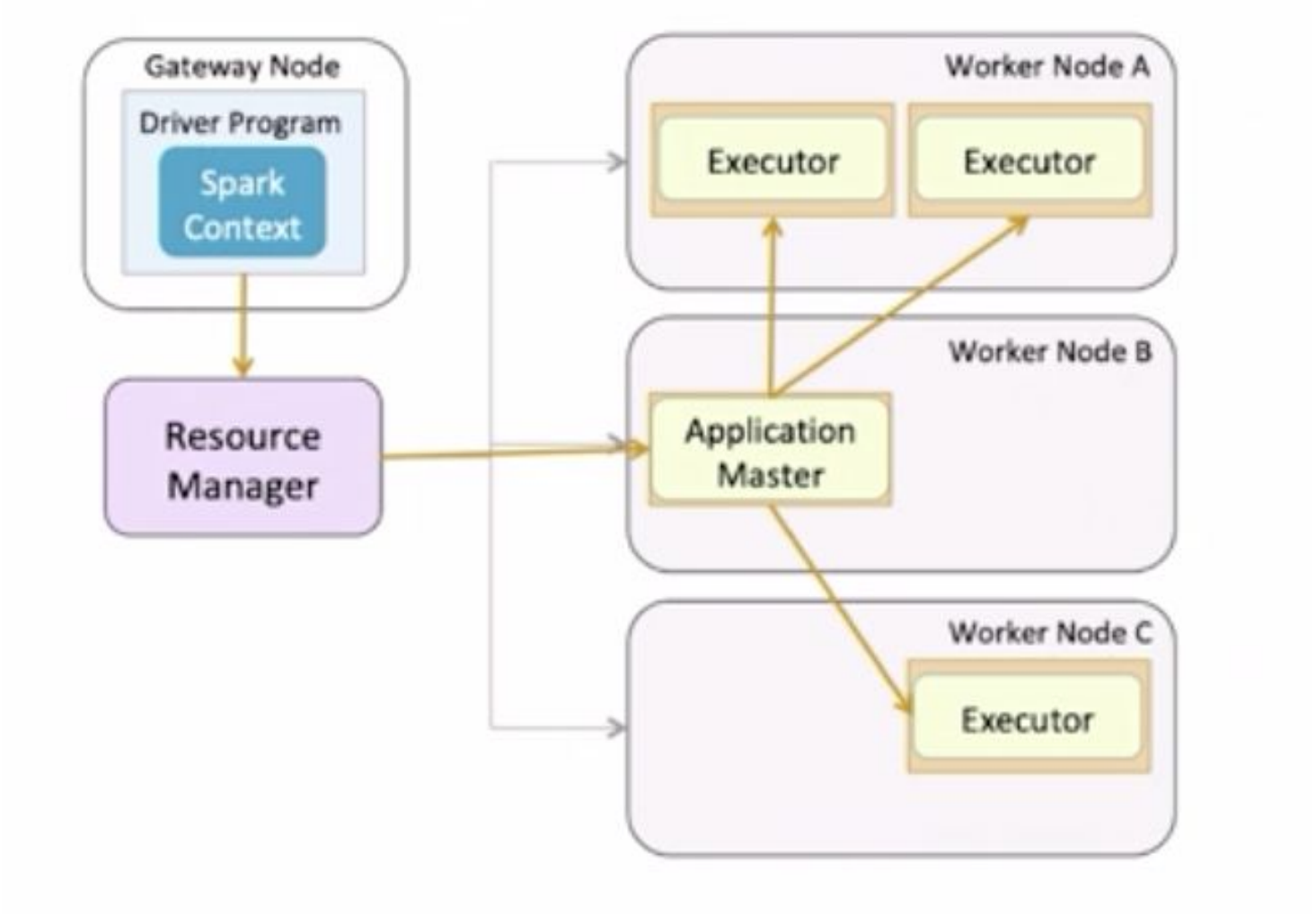
Spark Core Concepts , Internals & Architecture

Saurav Agarwal

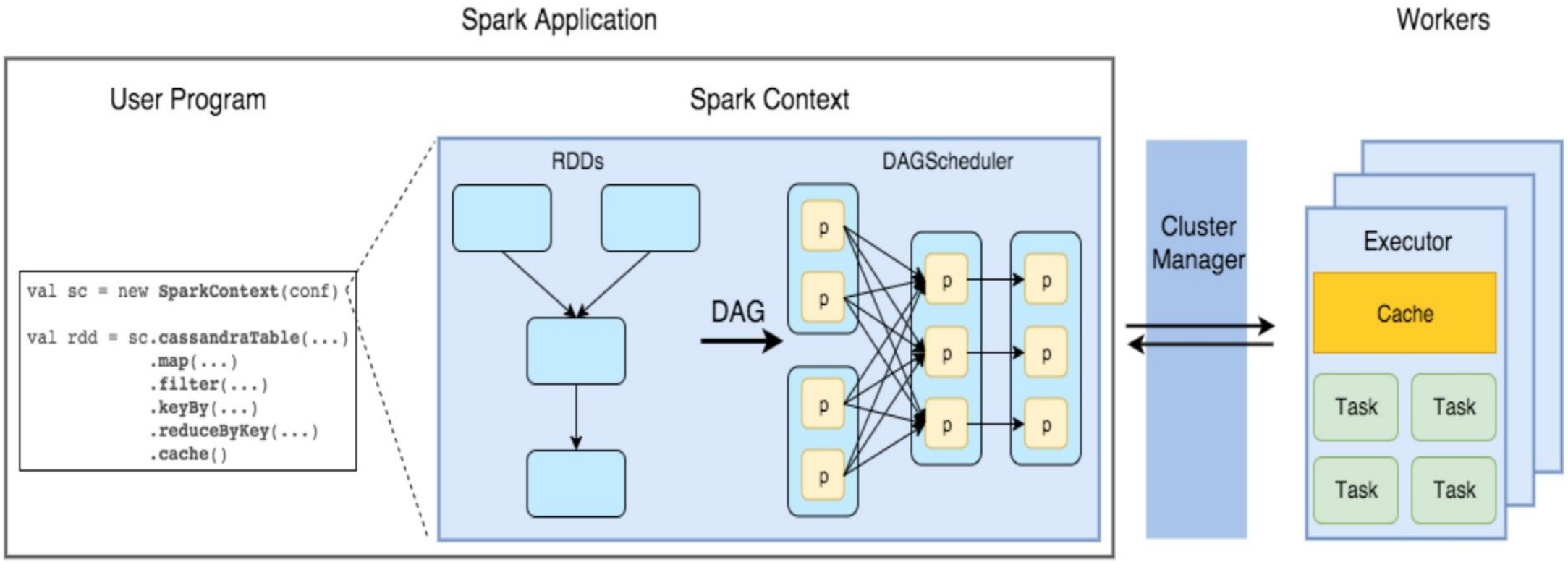
Spark Core Concepts

- Spark is built around the concepts of Resilient Distributed Datasets and Direct Acyclic Graph representing transformations and dependencies between them.
- Spark Application (often referred to as Driver Program or Application Master) at high level consists of SparkContext and user code which interacts with it creating RDDs and performing series of transformations to achieve final result.
- These transformations of RDDs are then translated into DAG and submitted to Scheduler to be executed on set of worker nodes.

Spark Execution Workflow



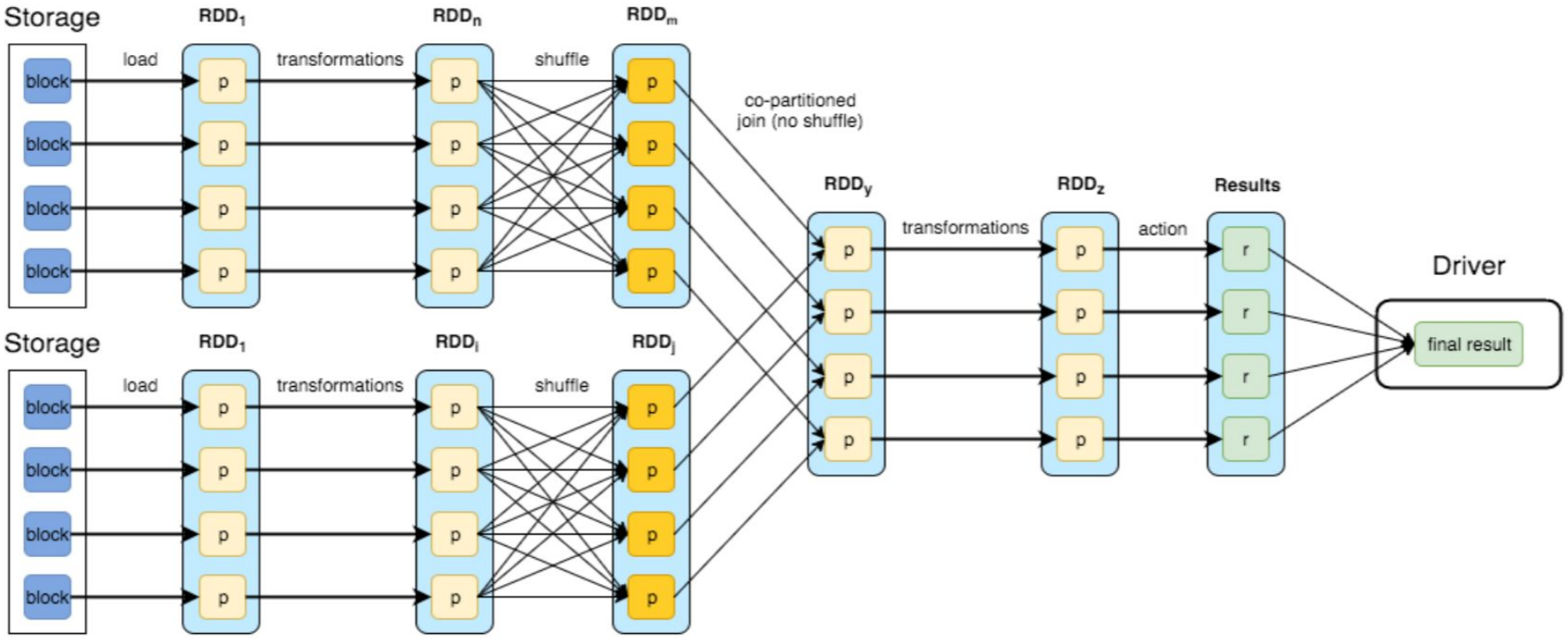
Spark Execution Workflow



Spark Execution Workflow

- User code containing RDD transformations forms Direct Acyclic Graph
- DAG then split into stages of tasks by DAGScheduler.
- Stages combine tasks which don’t require shuffling/repartitioning.
- Tasks run on workers and results then return to client.

Let Us Analyze a DAG

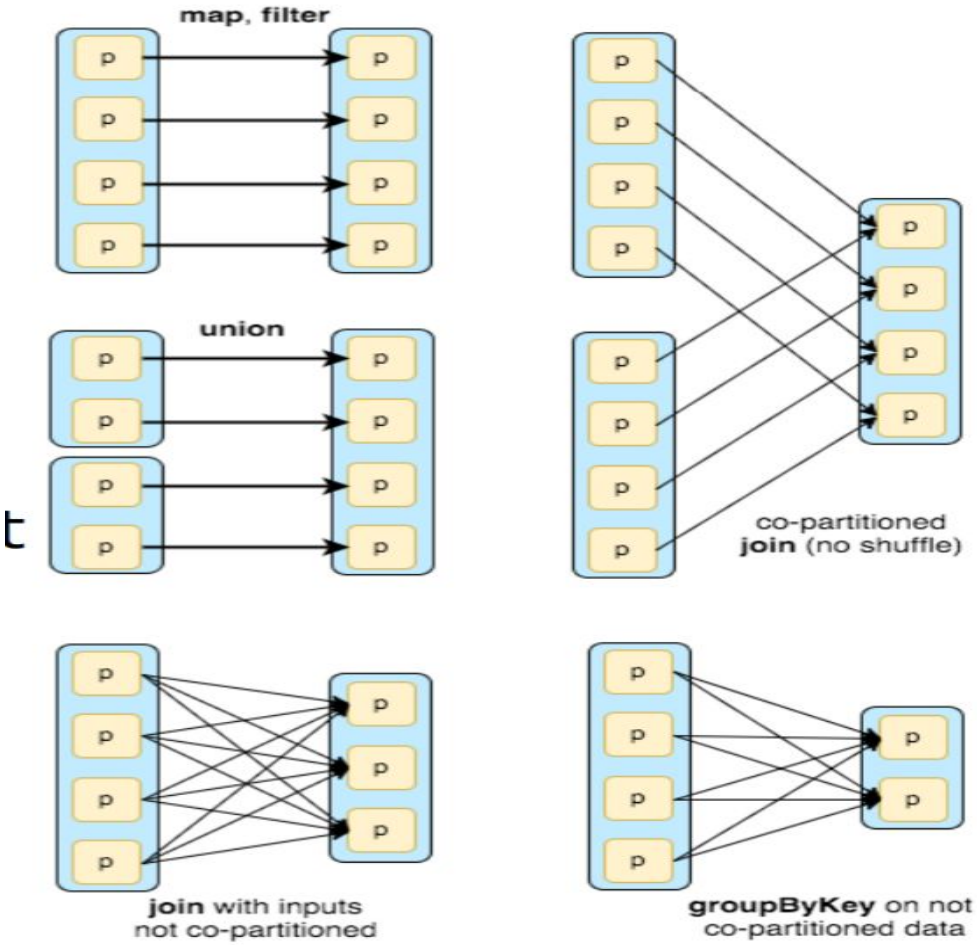


Let Us Analyze a DAG

- Any data processing workflow could be defined as reading the data source.
- Applying set of transformations and materializing the result in different ways.
- Transformations create dependencies between RDDs.
- The dependencies are usually classified as "narrow" and "wide".

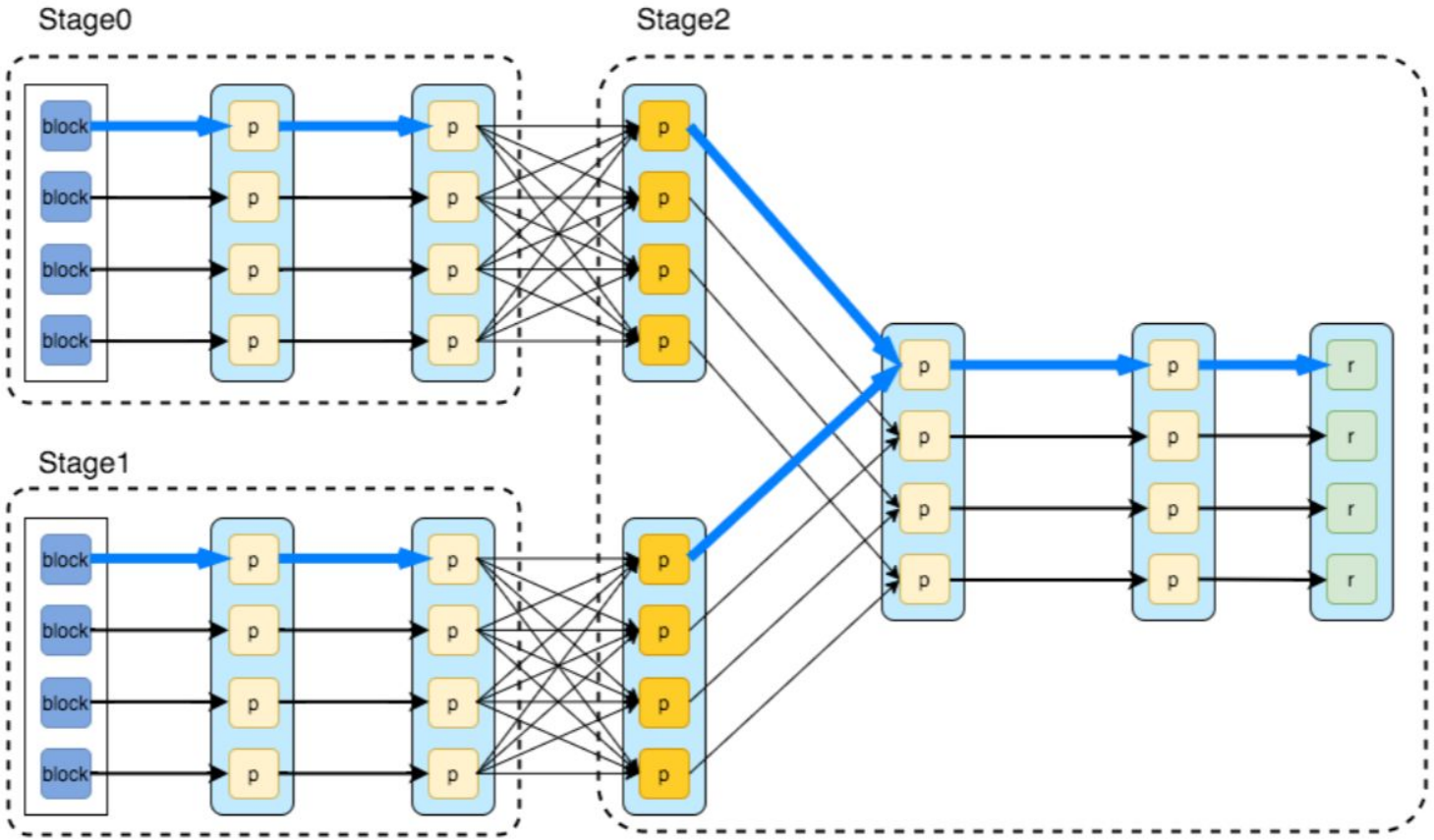
Let Us Analyze a DAG

- Narrow (pipelineable)
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed
- Wide (shuffle)
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - if some partition is lost from all the ancestors a complete recomputation is needed



Splitting DAG Into Stages

- Spark stages are created by breaking the RDD graph at shuffle boundaries



LIST OF NARROW VS WIDE TRANSFORMS

Transformations with *(usually)* Narrow dependencies:

- map
- mapValues
- flatMap
- filter
- mapPartitions
- mapPartitionsWithIndex

Transformations with (usually) Wide dependencies: (might cause a shuffle)

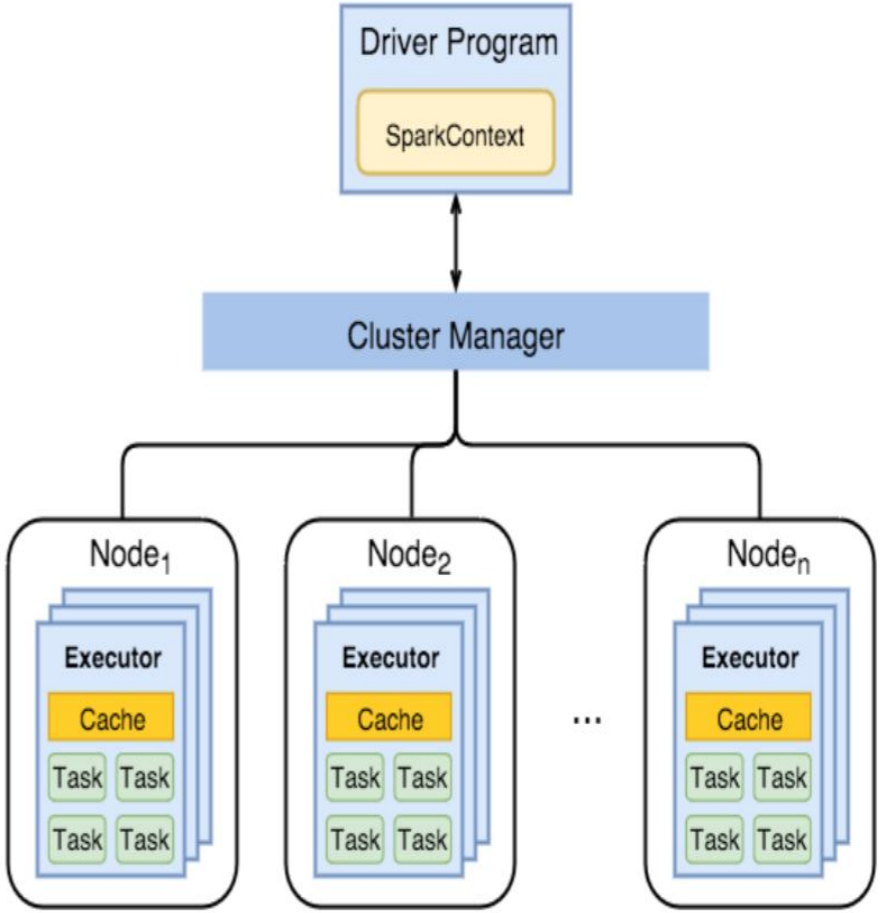
- cogroup
- groupWith
- join
- leftOuterJoin
- rightOuterJoin
- groupByKey
- reduceByKey
- combineByKey
- distinct
- intersection
- repartition
- coalesce

Splitting DAG Into Stages

- RDD operations with "narrow" dependencies, like map() and filter(), are pipelined together into one set of tasks in each stage operations with shuffle dependencies require multiple stages (one to write a set of map output files, and another to read those files after a barrier).
- In the end, every stage will have only shuffle dependencies on other stages, and may compute multiple operations inside it. The actual pipelining of these operations happens in the RDD.compute() functions of various RDDs

Spark Components

- From high level there are three major components.
- Spark driver
  - separate process to execute user applications
  - creates SparkContext to schedule jobs execution and negotiate with cluster manager

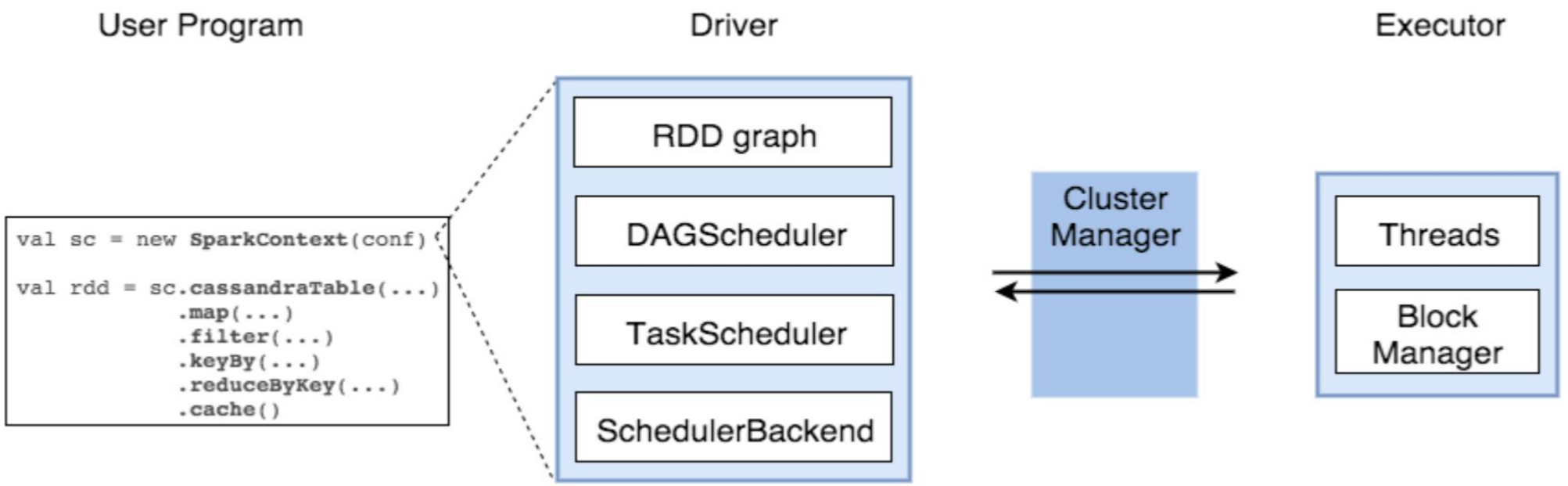


Spark Components

- Executors
  - Run tasks scheduled by driver
  - store computation results in memory, on disk or off-heap
  - interact with storage systems
- Cluster Manager
  - Mesos
  - YARN
  - Spark Standalone

Spark Components

- Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster:



Spark Components

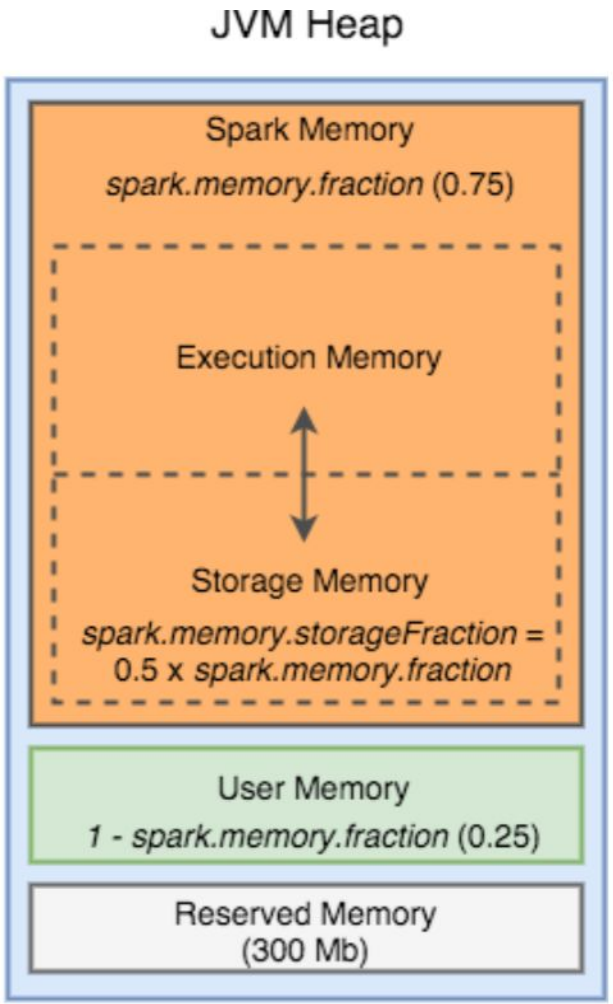
- SparkContext
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster
- DAGScheduler
  - computes a DAG of stages for each job and submits them to TaskScheduler
  - determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs
- TaskScheduler
  - responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers(slowness)

Spark Components

- SchedulerBackend – Resource Manager
  - backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)
- BlockManager
  - provides interfaces for putting and retrieving blocks both local and external

Memory Management

- Execution Memory
  - storage for data needed during tasks execution
  - shuffle-related data
- Storage Memory
  - storage of cached RDDs and broadcast variables
  - possible to borrow from execution memory (spill otherwise)
  - safeguard value is 50% of Spark Memory when cached blocks are immune to eviction
- User Memory
  - user data structures and internal metadata in Spark
  - safeguarding against OOM
- Reserved Memory
  - memory needed for running executor itself and not strictly related to Spark



Spark Dataframes

Saurav Agarwal

Dataframe

DataFrames introduced in Spark 1.3 as extension to RDDs

Distributed collection of data organized into named columns

» Equivalent to Pandas and R DataFrame, but distributed

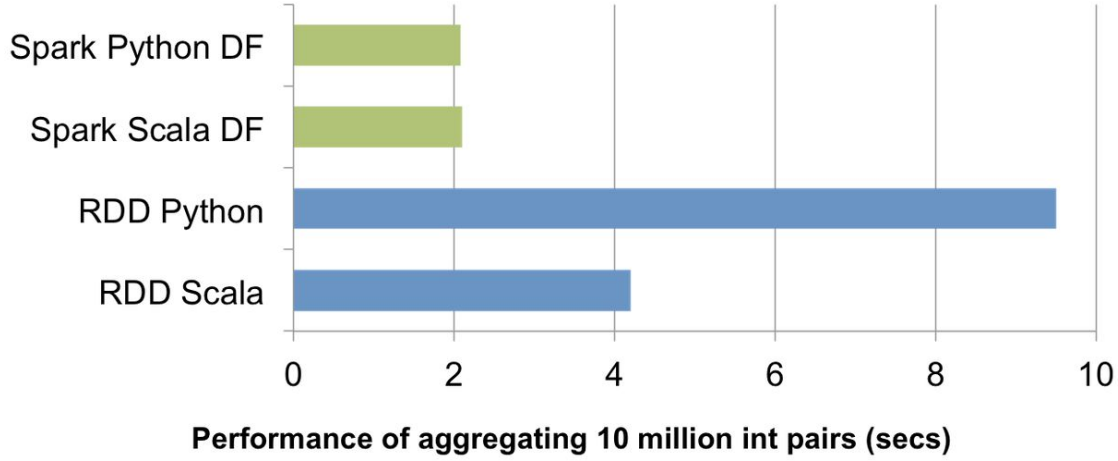
Types of columns inferred from values

Easy to convert between Pandas and pySpark

» Note: pandas DataFrame must fit in driver

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

Almost 5x pySpark performance on a single machine



CREATING DATAFRAMES

- Using an RDD spark.createDataFrame(some_rdd)

and then passing rdd.toDF(List of Columns)

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

SCHEMA- STRUCTURE OF DATA

- A schema is the description of the structure of your data (which together create a Dataset in Spark SQL). It can be implicit (and inferred at runtime) or explicit (and known at compile time).
- A schema is described using StructType which is a collection of StructField objects (that in turn are tuples of names, types, nullability classifier and metadata of key-value pairs).
- We can create Dataframe using StructType as well like the below

createDataFrame(another_rdd, schemaofstructtype)

STRUCTTYPE

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

```
schemaUntyped = new StructType()
 .add("a", "int")
.add("b", "string")
```

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

- // it is equivalent to the above expression
- import org.apache.spark.sql.types.{IntegerType, StringType}

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

READING DATA FROM EXTERNAL DATA SOURCE

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

QUERY and Processing DATAFRAME

Commonly used transformations are as below:

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

Writing Dataframes

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

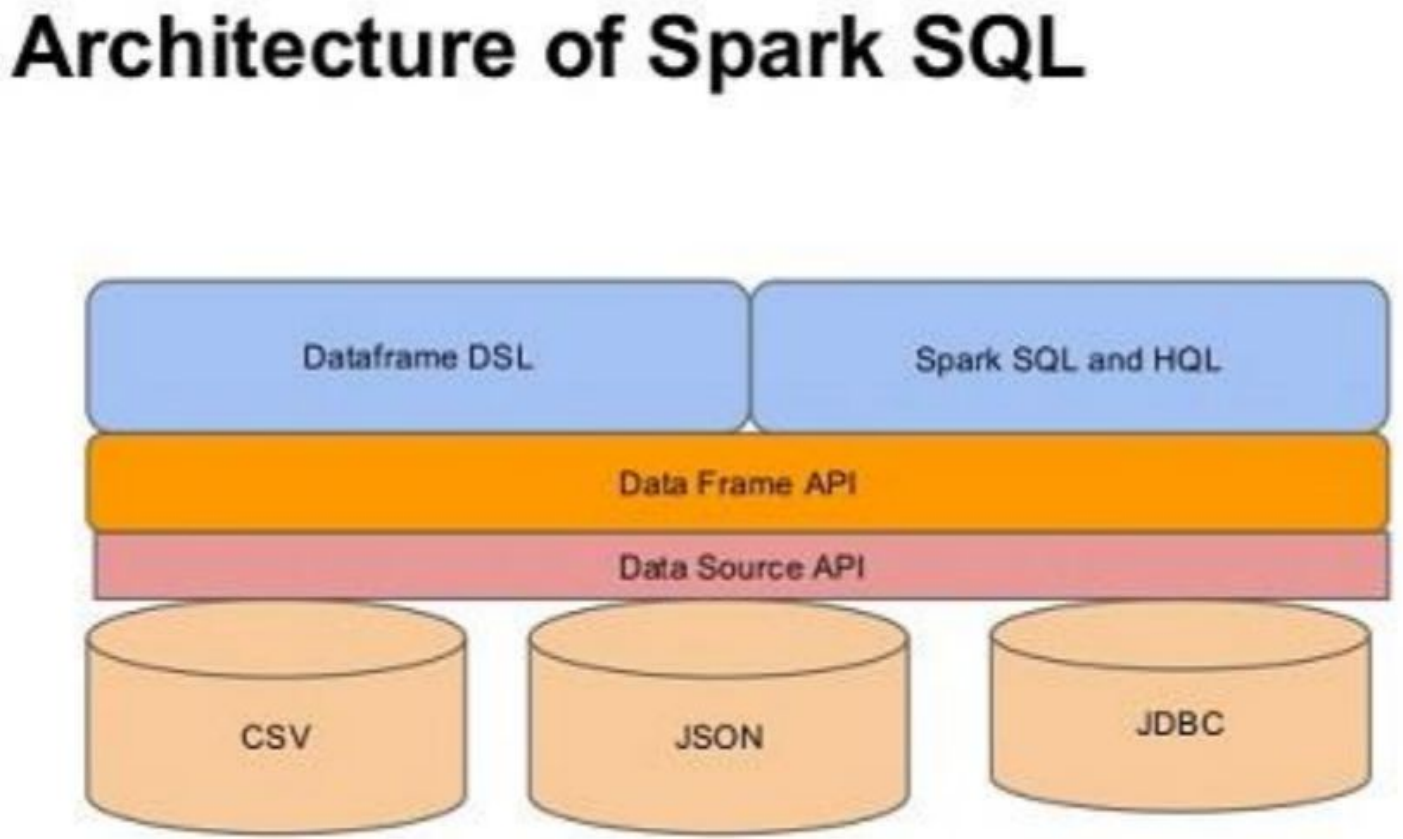
Spark SQL

Saurav Agarwal

Why Spark SQL?

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

Architecture of Spark SQL



SQL - A language for Relational DBs

SQL = Structured Query Language

Supported by pySpark DataFrames (SparkSQL)

Some of the functionality SQL provides:

» Create, modify, delete relations

» Add, modify, remove tuples

» Specify queries to find tuples matching criteria

- Register a DataFrame as a named temporary table to run SQL.
- df.registerTempTable("auctions") (1)
- spark.sql("SELECT count(*) AS count FROM auctions")

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

Queries in SQL

Single-table queries are straightforward

To find just names and logins:

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

SparkSQL and Spark DataFrames join() supports:

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

App deployment in PySpark

- **The Spark submit script provides many options to specify how the application should run**
  - Most are the same as for `pyspark` and `spark-shell`
- **General submit flags include**
  - `master`: `local`, `yarn`, or a Mesos or Spark Standalone cluster manager URI
  - `jars`: Additional JAR files
  - `pyfiles`: Additional Python files (Python only)
  - `driver-java-options`: Parameters to pass to the driver JVM
- **YARN-specific flags include**
  - `num-executors`: Number of executors to start application with
  - `driver-cores`: Number cores to allocate for the Spark driver
  - `queue`: YARN queue to run in
- **Show all available options**
  - `help`

```
$ spark-submit NameList.py people.json namelist/
```

Spark Dataframes - Catalyst & Tungsten Optimizers

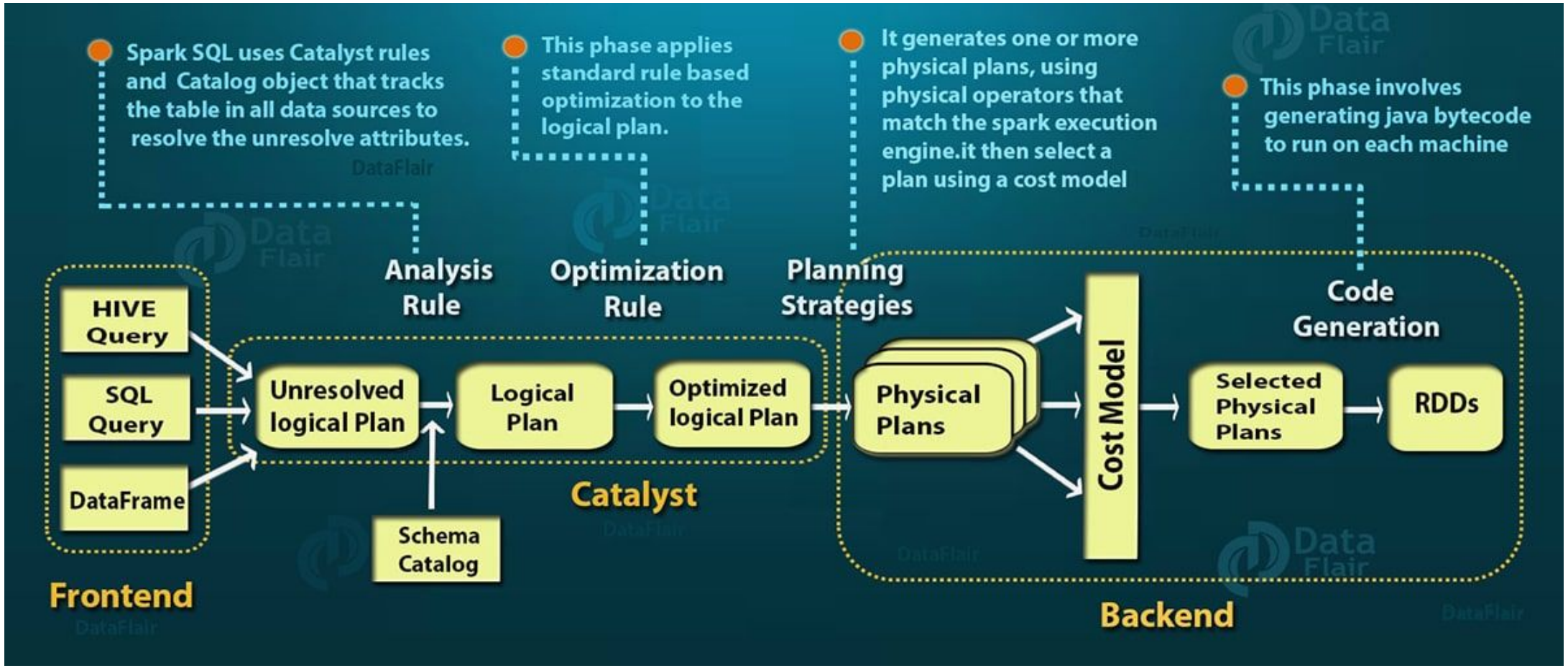
Saurav Agarwal

Catalyst Optimizer

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

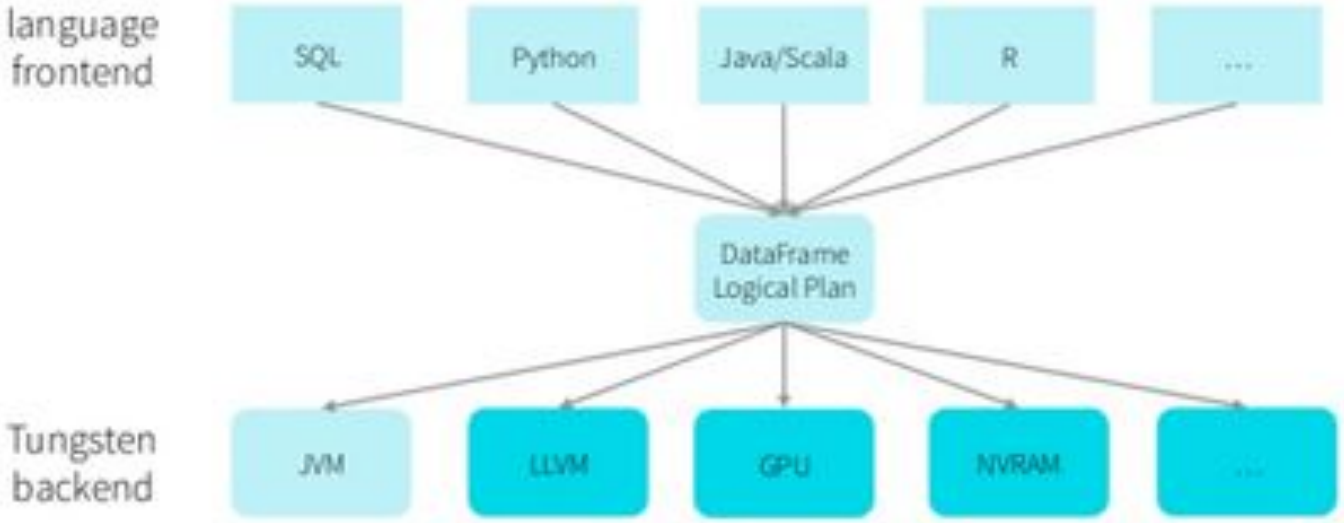
- Minimize data transfer between executors
  - Such as *broadcast* joins—small data sets are pushed to the executors where the larger data sets reside
- Minimize wide (shuffle) operations
  - Such as unioning two RDDs—grouping, sorting, and joining do not require shuffling
- Pipeline as many operations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

Catalyst Optimizer



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

Project Tungsten



Optimization Features

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

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

RDD vs Dataframe vs Dataset

Saurav Agarwal

RDD vs DataFrame vs Dataset

When to use RDD:

- you want low-level transformation and actions and control on your dataset;
- your data is unstructured, such as media streams or streams of text;
- you want to manipulate your data with functional programming constructs than domain specific expressions;
- you don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and
- you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.

DataFrames

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

Datasets

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

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

Parquet vs ORC vs Avro

Saurav Agarwal

ORC FORMAT

- ORC stands for Optimized Row Columnar which means it can store data in an optimized way than the other file formats. ORC reduces the size of the original data up to 75%(eg: 100GB file will become 25GB). As a result the speed of data processing also increases. ORC shows better performance than Text, Sequence and RC file formats.
  An ORC file contains rows data in groups called as Stripes along with a file footer. ORC format improves the performance when Hive is processing the data.
- An ORC file contains groups of row data called **stripes**, along with auxiliary information in a **file footer**. At the end of the file a **postscript** holds compression parameters and the size of the compressed footer.
- The default stripe size is 250 MB. Large stripe sizes enable large, efficient reads from HDFS.
- The file footer contains a list of stripes in the file, the number of rows per stripe, and each column's data type. It also contains column-level aggregates count, min, max, and sum.
- This diagram illustrates the ORC file structure:

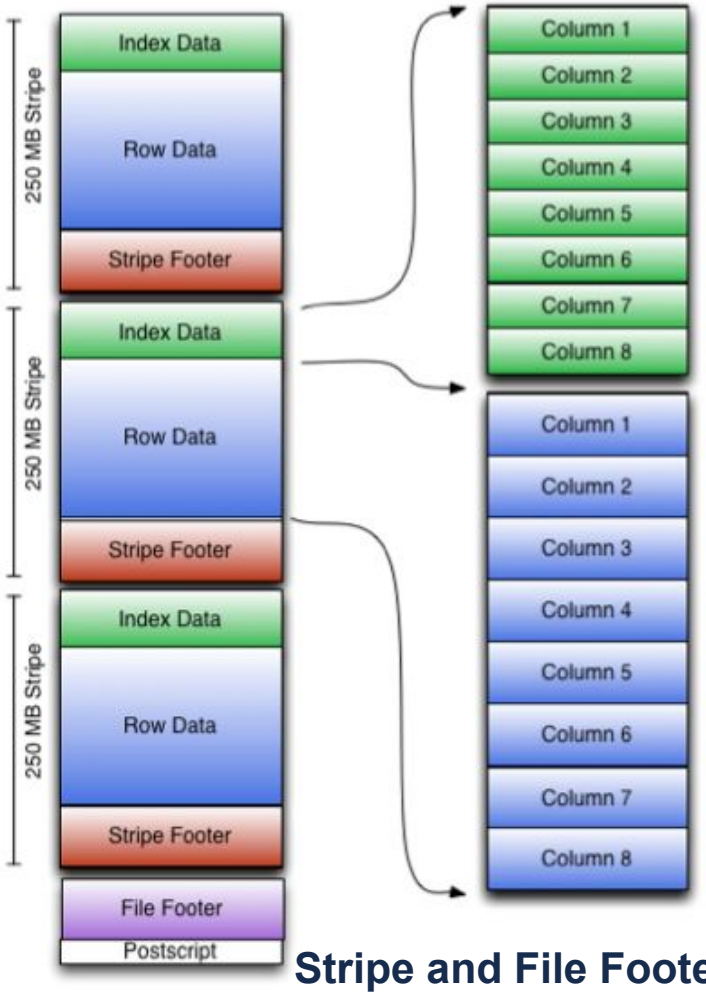
PARQUET FORMAT

- Parquet is an open source file format available to any project in the Hadoop ecosystem. Apache Parquet is designed for efficient as well as performant flat columnar storage format of data compared to row based files like CSV or TSV files.
- Columnar storage like Apache Parquet is designed to bring efficiency compared to row-based files like CSV. When querying, columnar storage you can skip over the non-relevant data very quickly. As a result, aggregation queries are less time consuming compared to row-oriented databases. This way of storage has translated into hardware savings and minimized latency for accessing data.
- Compression ratio is 60-70%.

AVRO FORMAT

- .avro
- Compression ratio 50-55%
- Schema is stored separately in avsc (avro schema) file
- Schema evolution advantage – Even if one column data doesn't appear automatically hive will default it to some value.
- The default value is specified in the avsc file.
- **Name , address, phone**
- 1,blr,99168
- 2,Noida, 9999
- 3,nocityfound,9777

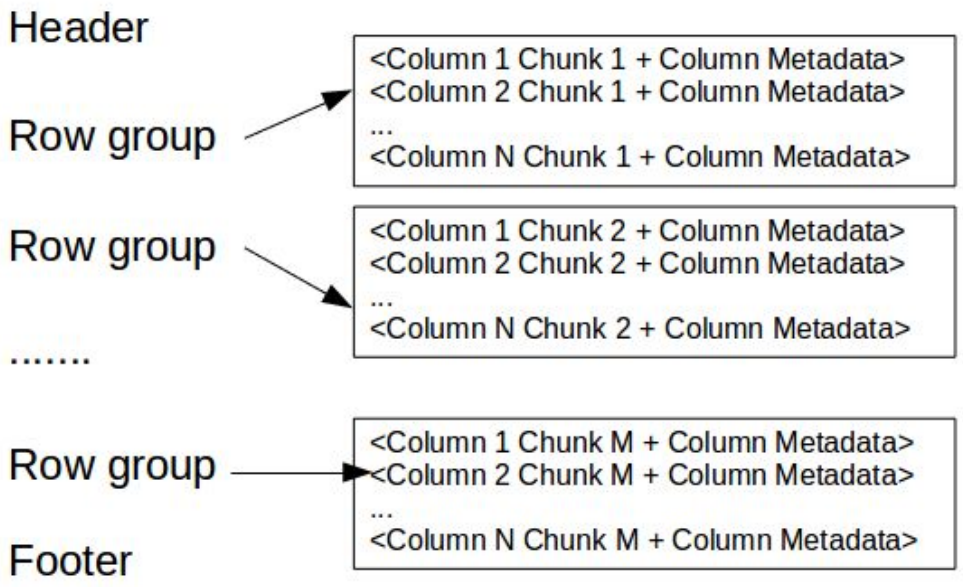
ORC



Index data includes min and max values for each column and the row positions within each column

Raw Data the row values

PARQUET



Stripe and File Footer contains the list of stripes(Files footer) in the file, the number of rows per stripe, and each column's data type. It also contains column-level aggregates count, min, max, and sum at respective levels.

Parquet vs Avro Format

- Avro is a row-based storage format for Hadoop.
- Parquet is a column-based storage format for Hadoop.

- If your use case typically scans or retrieves all of the fields in a row in each query, Avro is usually the best choice. If your dataset has many columns, and your use case typically involves working with a subset of those columns rather than entire records, Parquet is optimized for that kind of work.

Using UDFs with DataFrame

```
val data = List( List(0, "hello"),
 List(1, "world"))
val schema = StructType( Array( StructField("id", IntegerType, false),
StructField("word", StringType, false)))
val rdd = sparkContext.parallelize(data).map(Row.fromSeq(_))
val df1 = sqlContext.createDataFrame(rdd, schema)
def udfUpperCase=org.apache.spark.sql.functions.udf((word: String) => {
word.toUpperCase() })
 val upperUDF = df1.withColumn("uppercaseword",
udfUpperCase(df1("word")))
```

Using UDFs with SparkSQL

```
val data = List( List(0, "hello"), List(1, "world") )

val schema = StructType(Array( StructField("id", IntegerType, false),
StructField("word", StringType, false) ))

val rdd = sparkContext.parallelize(data).map(Row.fromSeq(_))

val df1 = sqlContext.createDataFrame(rdd, schema)

df1.createOrReplaceTempView("sample_data")

 sqlContext.udf.register("uppercaseword", (colA: String) => { colA.toUpperCase() })

sqlContext.sql("SELECT id, word, uppercaseword(word) FROM sample_data").show()
```

PARTITIONING

- Hive organizes tables horizontally into partitions.
- •It is a way of dividing a table into related parts based on the values of partitioned columns such as date, city, department etc.
- •Using partition, it is easy to query a portion of the data.
- •Partitioning can be done based on more than column which will impose multi-dimensional structure on directory storage.
- •In Hive, partitioning is supported for both managed and external tables.
- •The partition statement lets Hive alter the way it manages the underlying structures of the table’s data directory.
- •In case of partitioned tables, subdirectories are created under the table’s data directory for each unique value of a partition column.
- •When a partitioned table is queried with one or both partition columns in criteria or in the WHERE clause, what Hive effectively does is partition elimination by scanning only those data directories that are needed.
- •If no partitioned columns are used, then all the directories are scanned (full table scan) and partitioning will not have any effect.

CLASSIFICATION OF PARTITIONING

- •Static partitioning
- •Dynamic Partitioning
- **When to use static partitioning**
- •Static partitioning needs to be applied when we know data (supposed to be inserted) belongs to which partition.
- **When to use dynamic partitioning**
- •In static partitioning, every partitioning needs to be backed with individual hive statement which is not feasible for large number of partitions as it will require writing of lot of hive statements.
- •In that scenario dynamic partitioning is suggested as we can create as many number of partitions with single hive statement.

BUCKETING

- Bucketing concept is based on (hashing function on the bucketed column) mod (by total number of buckets). The hash_function depends on the type of the bucketing column.
- •Records with the same bucketed column will always be stored in the same bucket.
- •We use CLUSTERED BY clause to divide the table into buckets.
- •Physically, each bucket is just a file in the table directory, and Bucket numbering is 1-based.
- •Bucketing can be done along with Partitioning on Hive tables and even without partitioning.
- •Bucketed tables will create almost equally distributed data file parts, unless there is skew in data.
- •Bucketing is enabled by setting hive.enforce.bucketing= true;
- Advantages
- •Bucketed tables offer efficient sampling than by non-bucketed tables. With sampling, we can try out queries on a fraction of data for testing and debugging purpose when the original data sets are very huge.
- •As the data files are equal sized parts, map-side joins will be faster on bucketed tables than non-bucketed tables.
- •Bucketing concept also provides the flexibility to keep the records in each bucket to be sorted by one or more columns. This makes map-side joins even more efficient, since the join of each bucket becomes an efficient merge-sort.

BUCKETING VS PARTITIONING

- •Partitioning helps in elimination of data, if used in WHERE clause, where as bucketing helps in organizing data in each partition into multiple files, so that the same set of data is always written in same bucket.
- •Bucketing helps a lot in joining of columns.
- •Hive Bucket is nothing but another technique of decomposing data or decreasing the data into more manageable parts or equal parts.
- •Partitioning a table stores data in sub-directories categorized by table location, which allows Hive to exclude unnecessary data from queries without reading all the data every time a new query is made.
- •Hive does support Dynamic Partitioning (DP) where column values are only known at EXECUTION TIME. To enable Dynamic Partitioning :
- SET hive.exec.dynamic.partition =true;
- •Another situation we want to protect against dynamic partition insert is that the user may accidentally specify all partitions to be dynamic partitions without specifying one static partition, while the original intention is to just overwrite the sub-partitions of one root partition.
- SET hive.exec.dynamic.partition.mode =strict;
- To enable bucketing:

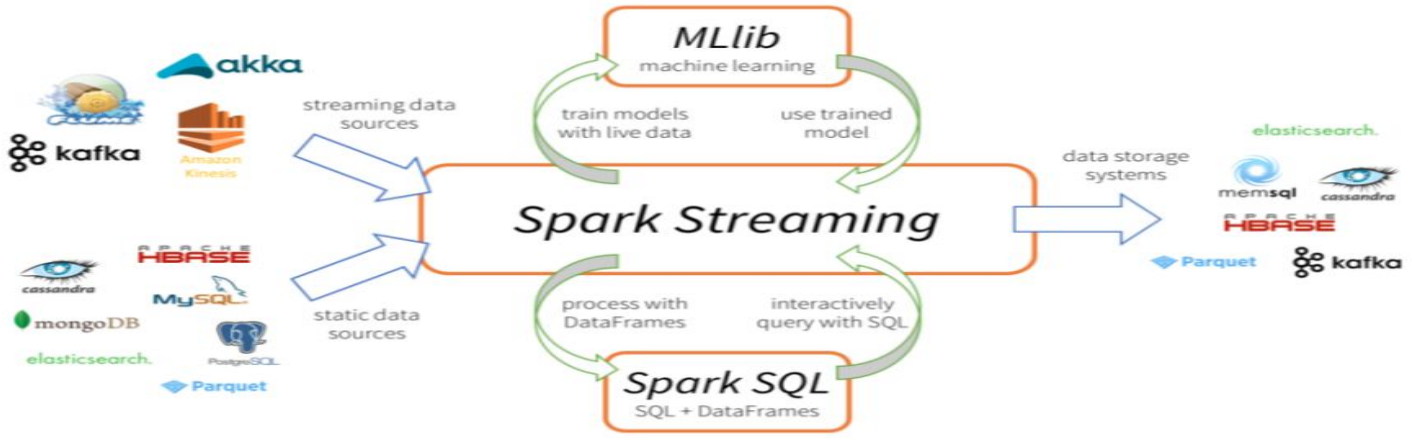
JOINS AND TYPES

Type	Approach	Pros	Cons
Shuffle Join	Join keys are shuffled using map/reduce and joins performed join side.	Works regardless of data size or layout.	Most resource-intensive and slowest join type.
Broadcast Join	Small tables are loaded into memory in all nodes, mapper scans through the large table and joins.	Very fast, single scan through largest table.	All but one table must be small enough to fit in RAM.
Sort-Merge-Bucket Join	Mappers take advantage of co-location of keys to do efficient joins.	Very fast for tables of any size.	Data must be sorted and bucketed ahead of time.

Spark Streaming

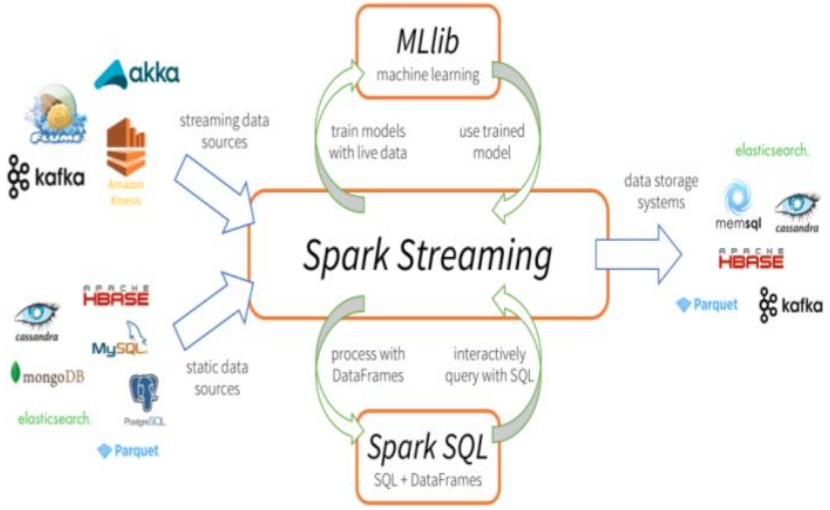
The Need for Streaming Architecture

- "Every company is now a software company" equates to companies collecting more data than ever and wanting to get value from that data in real-time.



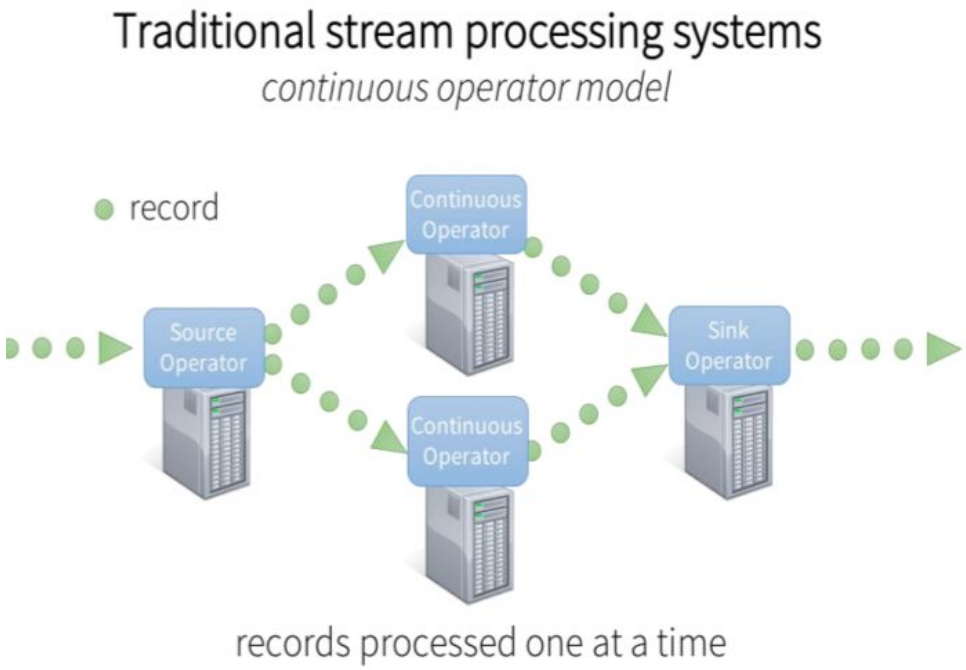
The Need for Streaming Architecture

- Sensors, IoT devices, social networks, and online transactions are all generating data that needs to be monitored constantly and acted upon quickly.
- Making a purchase online means that all the associated data (e.g. date, time, items, price) need to be stored and made ready for organizations to analyze and make prompt decisions based on the customer's behavior.
- Fraudulent bank transactions requires testing transactions against pre-trained fraud models as the transactions occur (i.e. as data streams) to quickly stop fraud in its track.



Stream Processing Architecture

- At high level, modern distributed stream processing pipelines execute as follows:
- Receive streaming data from data sources (e.g. live logs, system telemetry data, IoT device data, etc.) into some data ingestion system like Apache Kafka, Amazon Kinesis, etc.
- Process the data in parallel on a cluster. This is what stream processing engines are designed to do, as we will discuss this in detail in the next coming sessions.
- Output - The results out to downstream systems like HBase, Cassandra, Kafka, etc.



Challenges

- Continuous operators are a simple and natural model. However, with today's trend towards large-scale and more complex real-time analytics, this traditional architecture has also met some challenges. We designed Spark Streaming to satisfy the following requirements:

- Fast Failure and Straggler Recovery – With greater scale, there is a higher likelihood of a cluster node failing or unpredictably slowing down (i.e. Stragglers). The system must be able to automatically recover from failures and stragglers to provide results in real-time. Unfortunately, the static allocation of continuous operators to worker nodes makes it challenging for traditional systems to recover quickly from faults and stragglers.

- Load balancing – Uneven allocation of the processing load between the workers can cause bottlenecks in a continuous operator system. This is more likely to occur in large clusters and dynamically varying workloads. The system needs to be able to dynamically adapt the resource allocation based on the workload.

Challenges

- Unification of Streaming, Batch and Interactive Workloads – In many use cases, it is also attractive to query the streaming data interactively (after all, the streaming system has it all in memory), or to combine it with static datasets (e.g. pre-computed models). This is hard in continuous operator systems as they are not designed to the dynamically introduce new operators for ad-hoc queries. This requires a single engine that can combine batch, streaming and interactive queries.

- Advanced Analytics like Machine Learning and SQL Queries – More complex workloads require continuously learning and updating data models, or even querying the "latest" view of streaming data with SQL queries. Again, having a common abstraction across these analytic tasks makes the developer's job much easier.

Basic Sources

- In the session Spark Streaming, Stream from text data received over a TCP socket connection. Besides sockets, the StreamingContext API provides methods for creating Streams from files and Akka actors as input sources.
- • File Streams: For reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.)

Receiver Reliability

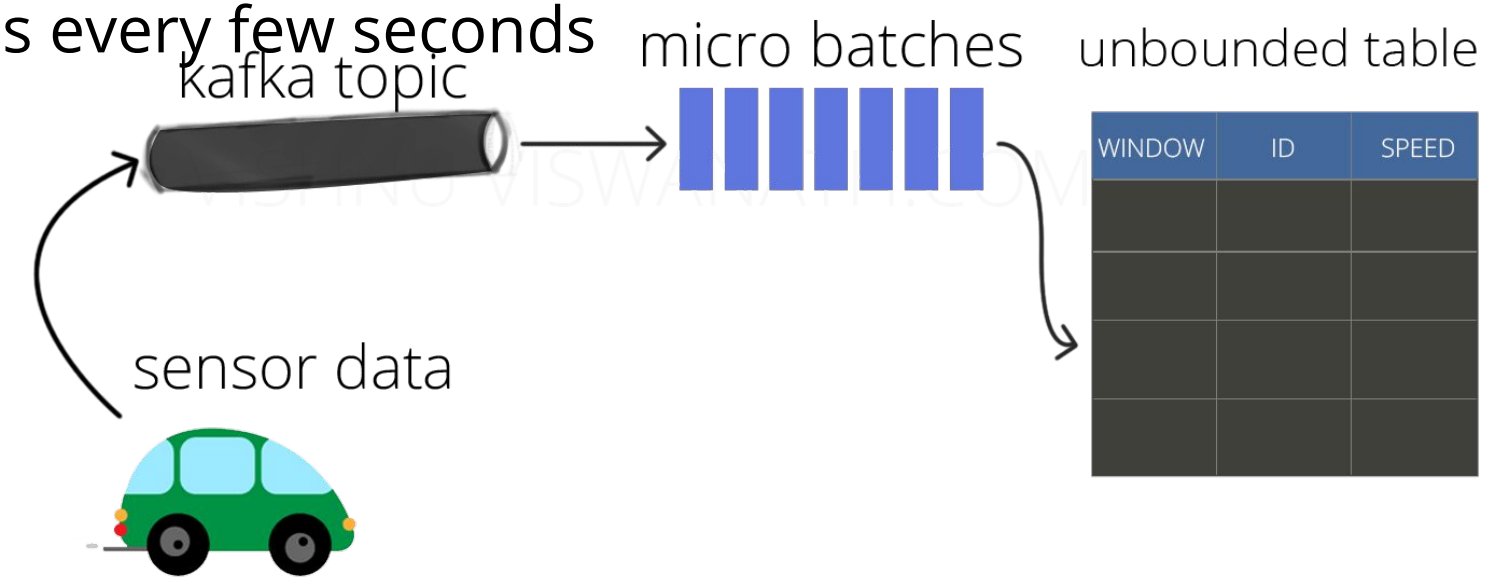
- There are two kinds of data sources based on their reliability.
- Sources like Kafka and Flume allow the transferred data to be acknowledged.
- If the system receiving data from these reliable sources acknowledges the received data correctly, it can be ensured that no data will be lost due to any kind of failure. This leads to two kinds of receiver:
  - Reliable Receiver – Correctly sends acknowledgment to a reliable source when the data has been received and stored in Spark with Replication
  - Unreliable Receiver – Doesn't send acknowledgement to a source. This can be used for sources that do not support acknowledgement, or even for reliable sources when one does not want or need to go into the complexity of acknowledgement.

Performance Tuning

- Getting the best performance out of a Spark Streaming application on a cluster requires a bit of tuning. At a high level, you need to consider two things:
  - Reducing the processing time of each batch of data by efficiently using the cluster resources
  - Setting the right batch size such that batches of data can be processed as fast as they are received (that is, data processing keeps up with the data ingestion)

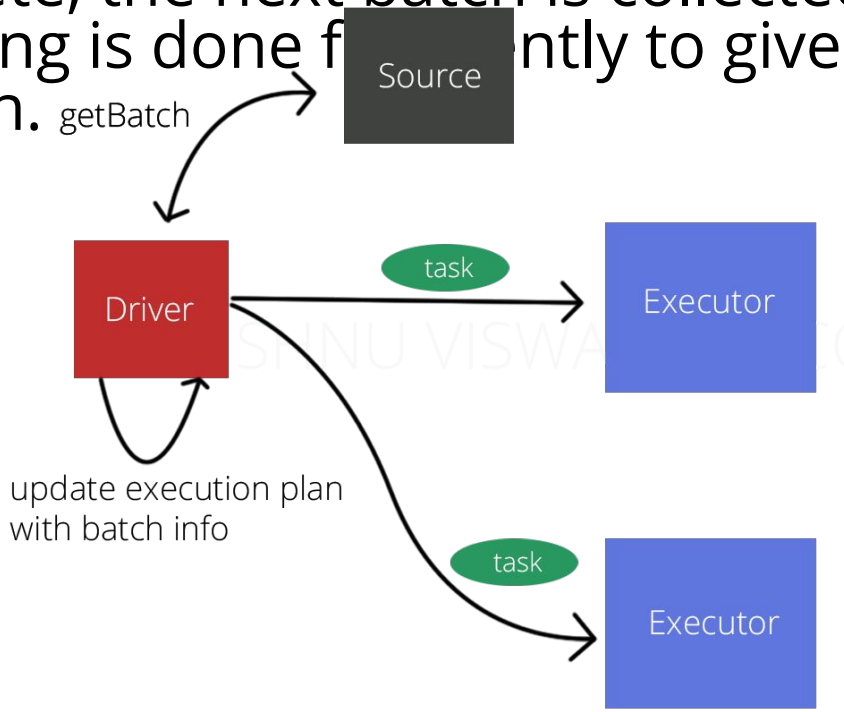
SPARK STRUCTURED STREAMING

- Structured Streaming is Apache Spark's streaming engine which can be used for doing near real-time analytics. Lets explore Structured Streaming by going through a very simple use case. Imagine you started a ride hailing company and need to check if the vehicles are over-speeding. We will create a simple near real-time streaming application to calculate the average speed of vehicles every few seconds



MICRO BATCH BASED STREAMING

- Structured Streaming in Spark, uses **micro-batching** to do streaming. That is, spark waits for a very small interval say 1 second (or even 0 seconds - i.e., as soon as possible) and batches together all the events that were received during that interval into a micro batch. This micro batch is then scheduled by the Driver to be executed as Tasks at the Executors. After a micro-batch execution is complete, the next batch is collected and scheduled again. This scheduling is done f ntly to give an impression of streaming execution.



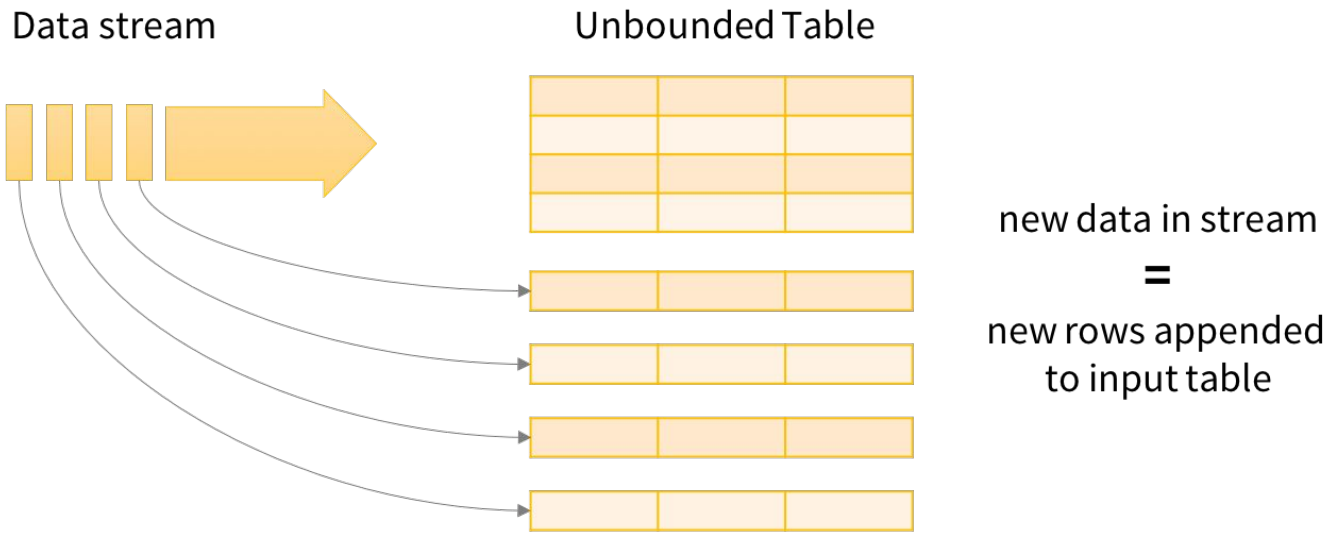
CONTINUOUS

- In version 2.3, Spark released a new execution engine called Continuous Processing, which does not do micro-batching. Instead, it launches long runnings tasks that read and process incoming data continuously.

```
.writeStream
    .format("kafka")
    .option("kafka.bootstrap.servers","localhost:9092")
    .option("topic", "fastcars")
    .option("checkpointLocation", "/tmp/sparkcheckpoint/")
    .queryName("kafka spark streaming kafka")
    .outputMode("update")
    .trigger(Trigger.Continuous("10 seconds")) //10 seconds is the
checkpoint interval.
    .start()
```

MODEL DETAILS

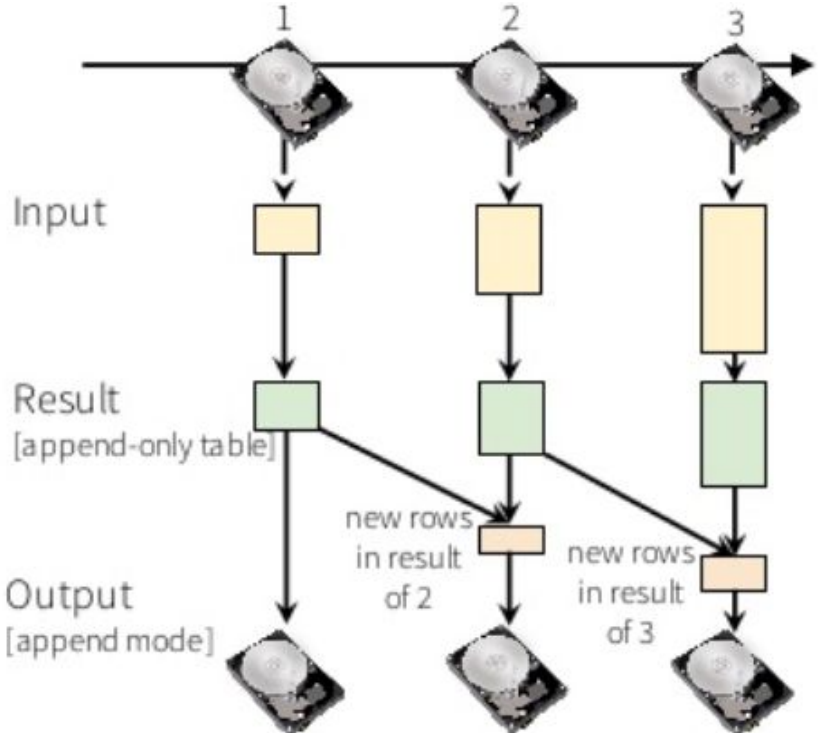
- Conceptually, Structured Streaming treats all the data arriving as an unbounded **input table**. Each new item in the stream is like a row appended to the input table. We won't actually retain all the input, but our results will be equivalent to having all of it and running a batch job.



Data stream as an unbounded Input Table

READ , PROCESS, WRITE

```
input = spark.read
      .format("json")
      .stream("source-path")

result = input
      .select("device", "signal")
      .where("signal > 15")

result.write
      .format("parquet")
      .startStream("dest-path")
```




```
input.groupBy(
        $"device-type",
        window($"event-time-col", "10 min"))
    .avg("signal")
```

```
input.avg("signal")
```

Continuously compute ***average*** signal ***across all devices***

Continuously compute ***average*** signal ***of each type of device*** in last 10 minutes using ***event-time***

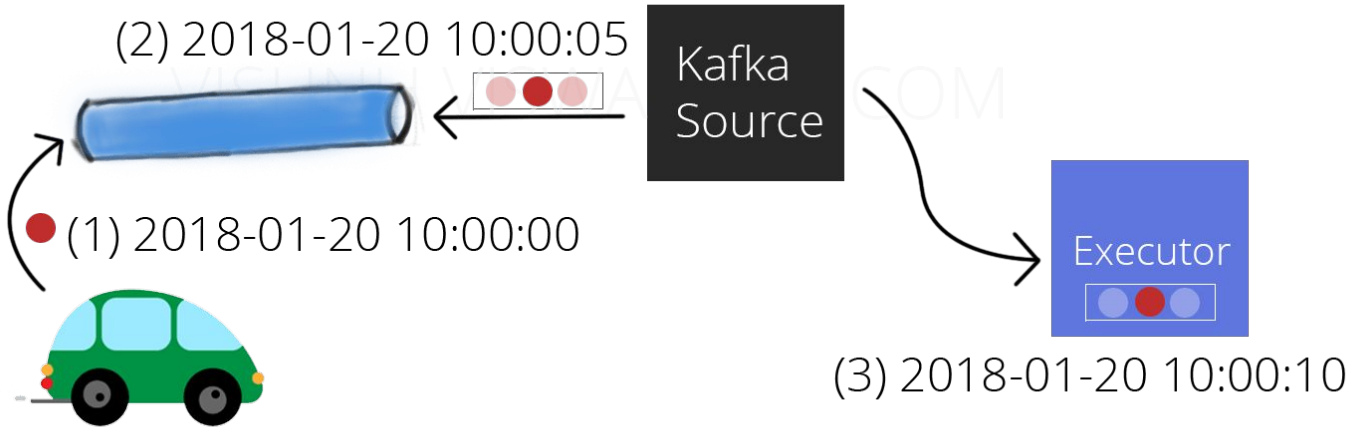
```
input.groupBy("device-type")
     .avg("signal")
```

Continuously compute ***average*** signal ***of each type of device***

EVENT TIME & PROCESSING TIME

EventTime is the time at which an event is generated at its source, whereas a ProcessingTime is the time at which that event is processed by the system. There is also one more time which some stream processing systems account for, that is IngestionTime - the time at which event/message was ingested into the System. It is important to understand the difference between EventTime and ProcessingTime.

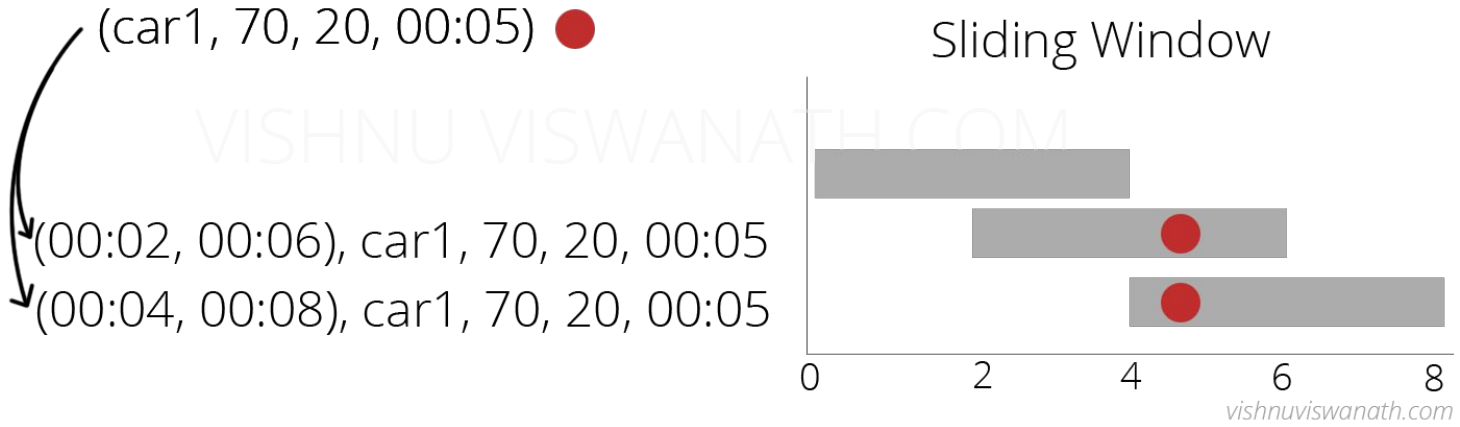
The red dot in the above image is the message, which originates from the vehicle, then flows through the Kafka topic to Spark's Kafka source and then reaches executor during task execution. There could be a slight delay (or maybe a long delay if there is any network connectivity issue) between these points. The time at the source is what is called an EventTime, the time at the executor is what is called the ProcessingTime. You can think of the ingestion time as the time at when it was first read into the system at the Kafka source (IngestionTime is not relevant for spark).



TUMBLING WINDOW & SLIDING WINDOW

A tumbling window is a non-overlapping window, that tumbles over every “window-size”. e.g., for a Tumbling window of size 4 seconds, there could be window for [00:00 to 00:04), [00:04: 00:08), [00:08: 00:12) etc (ignoring day, hour etc here). If an incoming event has EventTime 00:05, that event will be assigned the window - [00:04 to 00:08)

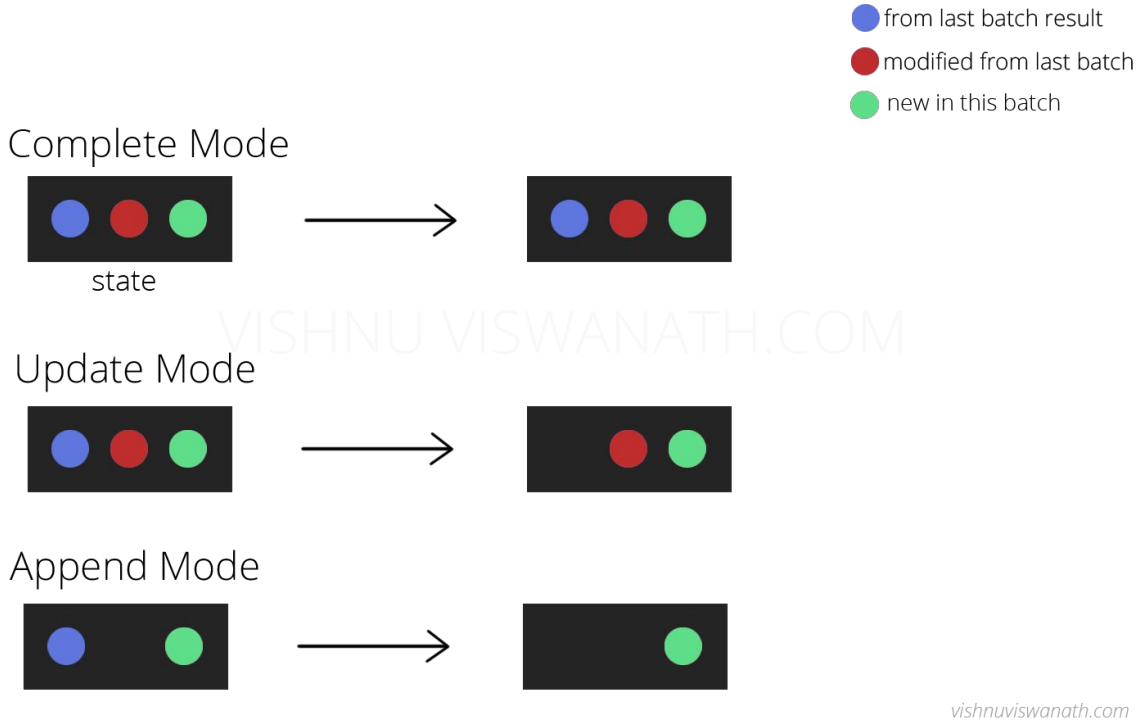
A SlidingWindow is a window of a given size(say 4 seconds) that slides every given interval (say 2 seconds). That means a sliding window could overlap with another window. For a window of size 4 seconds, that slides every 2 seconds there could windows [00:00 to 00:04), [00:02 to 00:06), [00:04 to 00:08) etc. Notice that the windows 1 and 2 are overlapping here. If an event with EventTime 00:05 comes in, that event will belong to the windows [00:02 to 00:06) and [00:04 to 00:08).



OUTPUT MODES

The last part of the model is **output modes**. Each time the result table is updated, the developer wants to write the changes to an external system, such as S3, HDFS, or a database. We usually want to write output incrementally. For this purpose, Structured Streaming provides three output modes:

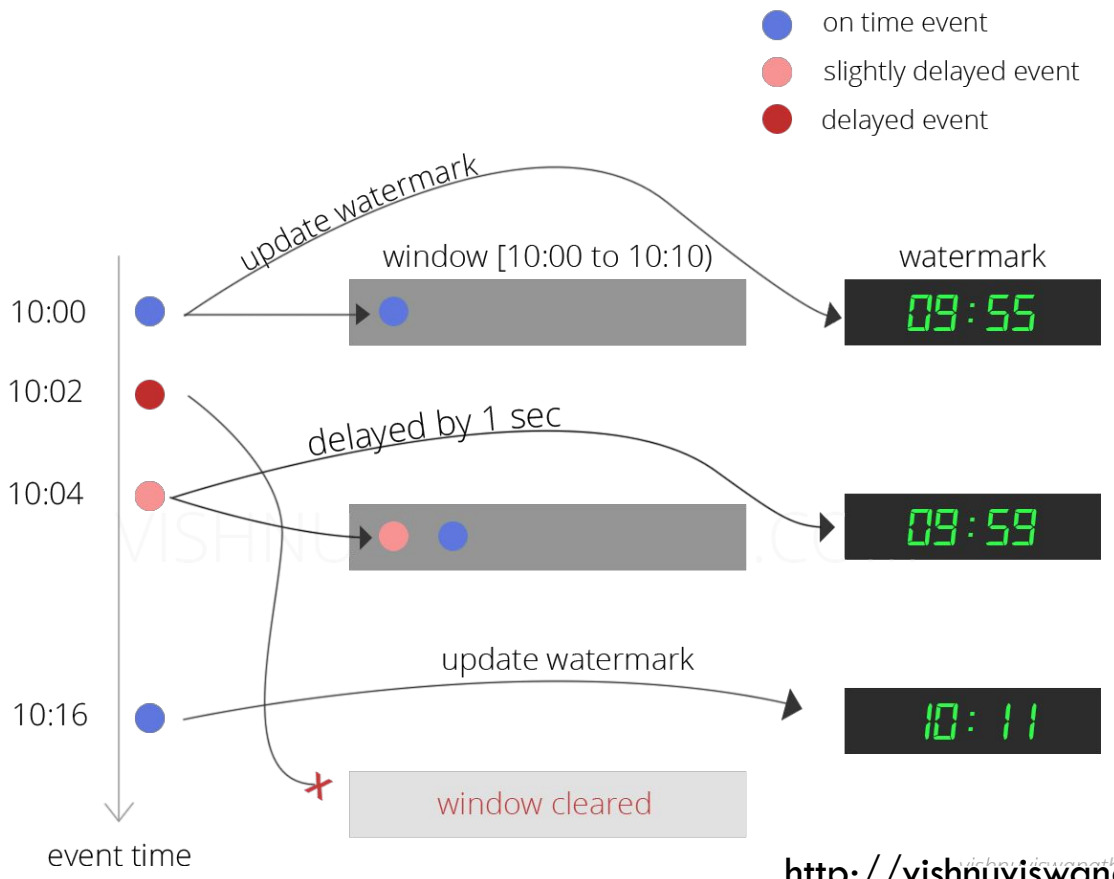
- **Append:** Only the new rows appended to the result table since the last trigger will be written to the external storage. This is applicable only on queries where existing rows in the result table cannot change (e.g. a map on an input stream).
- **Complete**: The entire updated result table will be written to external storage.
- **Update**: Only the rows that were updated in the result table since the last trigger will be changed in the external storage. This mode works for output sinks that can be updated in place, such as a MySQL table.



WATERMARK

In Spark, Watermark is used to decide when to clear a state based on current maximum event time. Based on the **delay** you specify, Watermark lags behind the maximum event time seen so far. e.g., if dealy is 3 seconds and current max event time is 10:00:45 then the watermark is at 10:00:42. This means that Spark will keep the state of windows who's end time is less than 10:00:42.

Watermark delay is 5 seconds here



http://vishnuviswanath.com/spark_structured_streaming.html

Spark MLLib machine learning

- Connect to batch/ real time streaming sources
- Data to be cleansed and transformed into a stream, stored in memory
- Models can or cannot be built in real time depending on the requirement and availability of libraries
- Do predictions in batch/real time
- Receive, store and communicate with external system

Spark ML Lib

- **Packages** – spark.mllib, spark.ml
- **Data types** – Local vector, Labelled point
- **Local vector** – vector of double values – Dense and Sparse
- e.g, a vector (1.0, 0.0, 3.0) can be represented in dense format as [1.0, 0.0, 3.0] or in sparse format as (3, [0, 2], [1.0, 3.0])
- **Labeled point** – contains the target variable(outcome) and list of features (predictors) ()
- **Process** - Load data into RDD **->** Transform RDD – Filter, Type conversion, centering, Scaling, etc. **->** Convert to labeled point **->** Split training and testing **->** Create model -> Tune -> Perform predictions

ML Pipelines

- Standard APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow.
- Inspired from SciKit-learn where multiple libraries of pre-processing, feature engineering and modelling are combined to give an output in using single line of code.
- Three major parts:

1. **DataFrame** – to hold the dataset
2. **Transformer** – Apply Functions/Models to datasets (may add columns), .transform()
3. **Estimator** – Converts a dataset into a Transformer, e.g. training the datasets using .fit()

- Sometimes pipelines may also contain params which are the args passed to a transformer.

Feature engineering in Spark

- **Extraction**: Extracting features from “raw” data.
- TF-IDF - Feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus.
- **TF**: Both HashingTF and CountVectorizer can be used to generate the term frequency vectors.
- HashingTF is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors.
- **IDF** is an Estimator which is fit on a dataset and produces an IDFModel. The IDFModel takes feature vectors (generally created from HashingTF or CountVectorizer) and scales each column.
- Word2Vec -Estimator which takes sequences of words representing documents and trains a Word2VecModel. The model maps each word to a unique fixed-size vector. The Word2VecModel transforms each document into a vector using the average of all words in the document
- CountVectorizer -CountVectorizer and CountVectorizerModel aim to help convert a collection of text documents to vectors of token counts.

https://spark.apache.org/docs/2.4.0/ml-features.html#pca

Transformation: Scaling, converting, or modifying features

- Tokenizer
- StopWordsRemover
- n-gram
- Normalizer
- StandardScaler
- Bucketizer
- SQLTransformer
- Imputer

Selection: Selecting a subset from a larger set of features

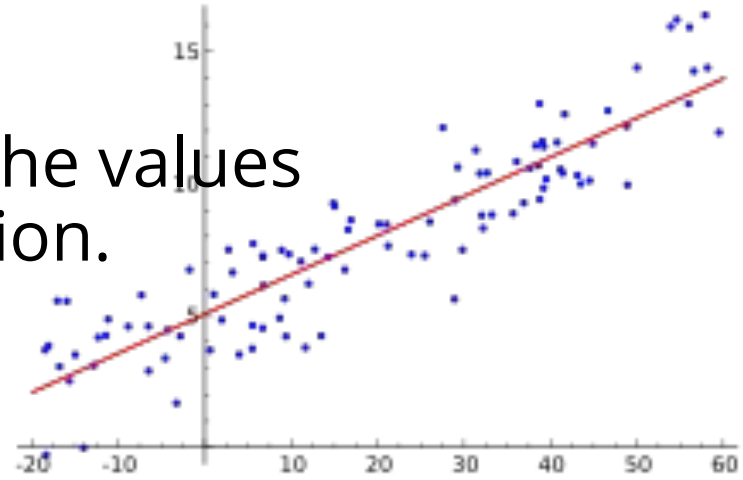
- VectorSlicer - Transformer that takes a feature vector and outputs a new feature vector with a sub-array of the original features. It is useful for extracting features from a vector column.
- Rformula - selects columns specified by an R model formula. Currently supports a limited subset of the R operators, including ‘~’, ‘.’, ‘:’, ‘+’, and ‘-‘.
- ChiSqSelector - It operates on labeled data with categorical features. ChiSqSelector uses the Chi-Squared test of independence to decide which features to choose.

More..

- **Correlation**
- Correlation is any of a broad class of statistical relationships involving dependence, though in common usage it most often refers to how close two variables are to having a linear relationship with each other.
- Correlation computes the correlation matrix for the input Dataset of Vectors using the specified method. The output will be a DataFrame that contains the correlation matrix of the column of vectors.

- **Hypothesis testing**
- Determine whether a result is statistically significant, whether this result occurred by chance or not.

Linear regression



- Estimate value of dependent variables from the values of independent variables with some correlation.
- Draw the best line to fit plotted points.

Example -

- Convert input string to vector & drop columns which are not needed
- Create rDD vectors.dense(atList(x).toFloat)
- Create labelled vectors & drop low correlation vectors (should be continuous)
- Run LR model on training data Print coefficients, intercepts, summary, lr.fit(trainingdata)
- Predict
- Evaluate using MSE (average((predicted value – actual value) ** 2, should be low), R square should be high as close to 1 as possible

K-Means Clustering

- Attempts to split data into K- groups that are closest to K centroids
- Un Supervised learning – uses only position of each data point
- Randomly pick K centroids -> Assign each data point to the centroid it’s closest to
- Re-compute centroids based on average position of each centroid’s points
- Iterate until points stop changing assignment to centroids
- If you want to predict the cluster for new points, just find the centroid they are closest to

Classification

- **Binary Classification** is the task of predicting a binary label. E.g., is an email spam or not spam?
- Supervised learning – uses only position of each data point
- For binary classification problems, the algorithm outputs a binary logistic regression model. Given a new data point, denoted by xx, the model makes predictions by applying the logistic function

Model selection and hyper parameter tuning

- An important task in ML is *model selection*, or using data to find the best model or parameters for a given task.
- **Estimator**: algorithm or Pipeline to tune
- Set of **ParamMaps** : parameters to choose from, sometimes called a “parameter grid” to search over

-Split the input data into separate training and test datasets.

-For each (training, test) pair, they iterate through the set of ParamMaps

-Identify the best ParamMap, CrossValidator finally re-fits the Estimator using the best ParamMap and the entire dataset.

- **Evaluator**: metric to measure how well a fitted Model does on held-out test data i.e evaluate the Model’s performance using the Evaluator.
- RegressionEvaluator , a BinaryClassificationEvaluator for binary data, or a MulticlassClassificationEvaluator

Spark Streaming

Use Cases:

- **Many big data applications need to process large data streams in real time, such as**
  - Continuous ETL
  - Website monitoring
  - Fraud detection
  - Advertisement monetization
  - Social media analysis
  - Financial market trends
  - Event-based data

Spark Streaming v/s DStreams

Structured Streaming	DStreams API
▪ DataFrame/Dataset-based ▪ Higher level API ▪ Best for structured or semi-structured data ▪ Provides SQL-like semantics ▪ Guarantees consistency between streaming and static queries ▪ Queries optimized by the Catalyst optimizer	▪ RDD-based ▪ Lower level API ▪ Best for unstructured data

- **Designed for end-to-end, continuous, real-time data processing**
- **Ensures consistency**
  - Guarantees exactly-once handling
  - Query results are the same on static or streaming data
- **Built-in support for time-series data**
  - Handles out-of-order and late events

Thank you!

Saurav Agarwal