Session 2

BigData Programming Models

Big Data Analytics Technology, MSc in Data Science,
Coventry University UK

Miyuru Dayarathna

Presentation Outline

- Introduction
- MapReduce
- Hive
- Spark
- Conclusion

Programming Model

- Different hardware platforms can be programmed in different ways exploiting their unique hardware features to develop efficient software.
- A programming model is a unique way of orchestrating computer hardware (execution model) coupled to an API or a particular pattern of code.

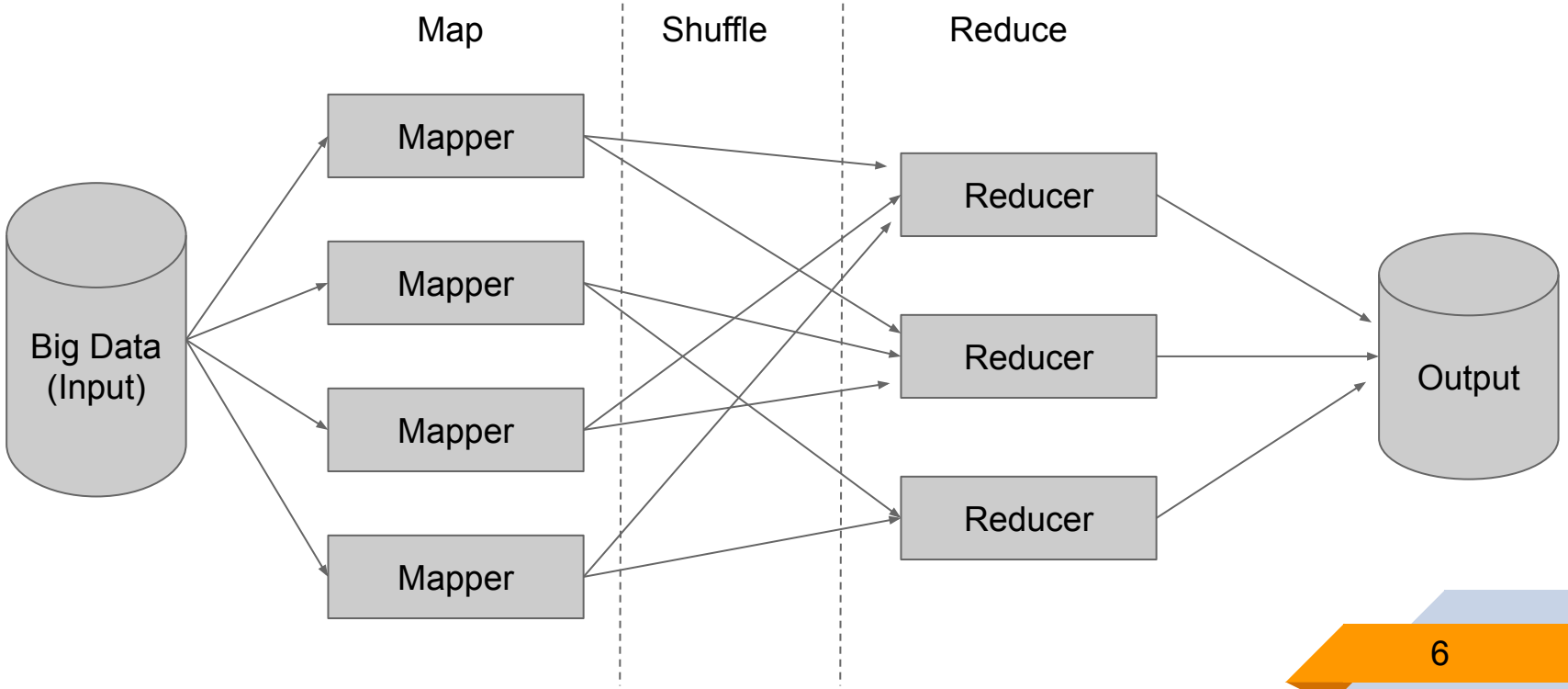
Programming Models

- POSIX Threads (pthread)
- OpenMP
- MPI
- MapReduce
- Bulk Synchronous Parallel (BSP)
  - Pregel
- Partitioned Global Address Space (PGAS)

MapReduce

- MapReduce is a programming model for expressing distributed computations on massive amounts of data
- It provides an execution framework for large-scale data processing on clusters of commodity servers.
- Originally developed by Google based on well-known principles in parallel and distributed processing by then.
- Widely adopted via an open-source implementation called Hadoop whose development was led by Yahoo!

MapReduce


Map
Shuffle
Reduce
Big Data
(Input)
Mapper
Mapper
Mapper
Mapper
Reducer
Reducer
Reducer
Output

MapReduce

- MapReduce works by dividing the processing into two phases: Map phase and Reduce Phase
  - similar to a Unix pipeline

    command_1 | command_2 | command_3 | .... | command_N

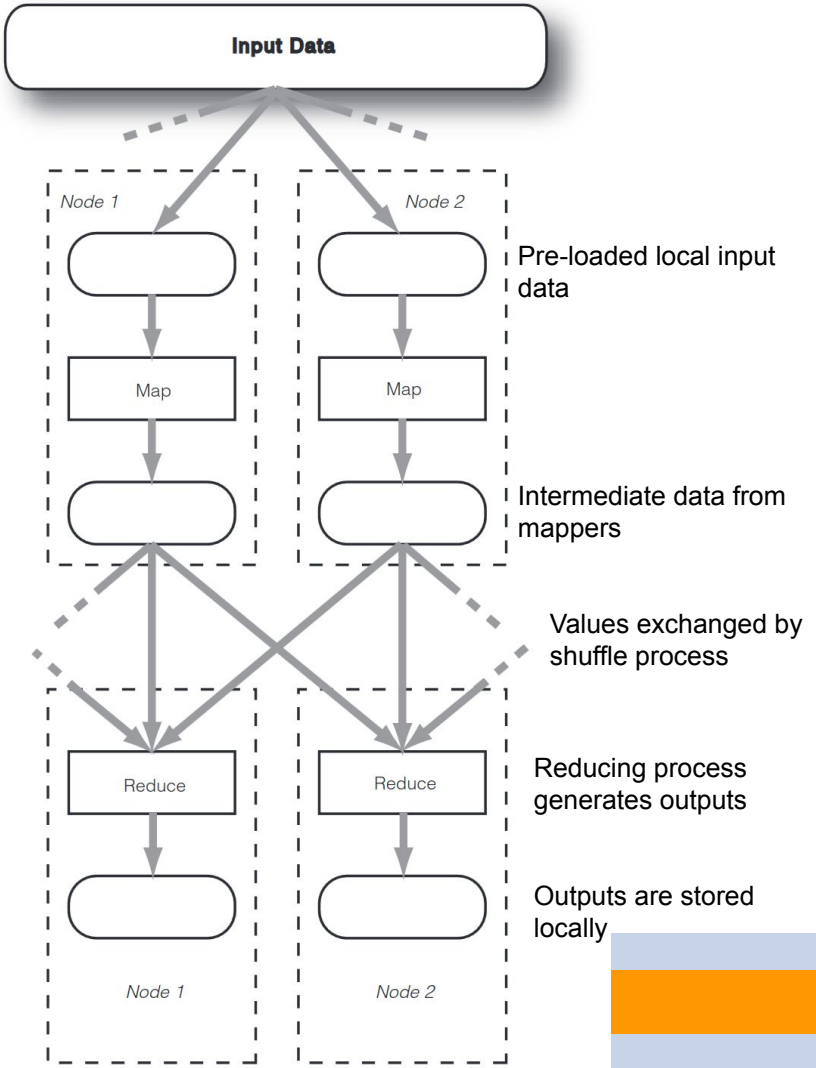
    cat result.txt | grep "Sachin Tendulkar" | tee file2.txt | wc -l

    Input | **Map** | Shuffle & Sort | **Reduce** | **Output**

- Each phase has key-value pair as input/output
- The programmer also specifies two functions: map function and reduce function

MapReduce

- In the general MapReduce data flow, after distributing input data to different nodes, the only time the nodes communicate with each other is at the "Shuffle" step.
- Idempotency is a key requirement in MapReduce



Source : Chuck Lam. "Hadoop in Action" (2011).

MapReduce - Hadoop - Introduction

- The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.
- Designed to scale up from single servers to thousands of machines, each offering local computation and storage.
- Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

https://hadoop.apache.org/

MapReduce - Hadoop

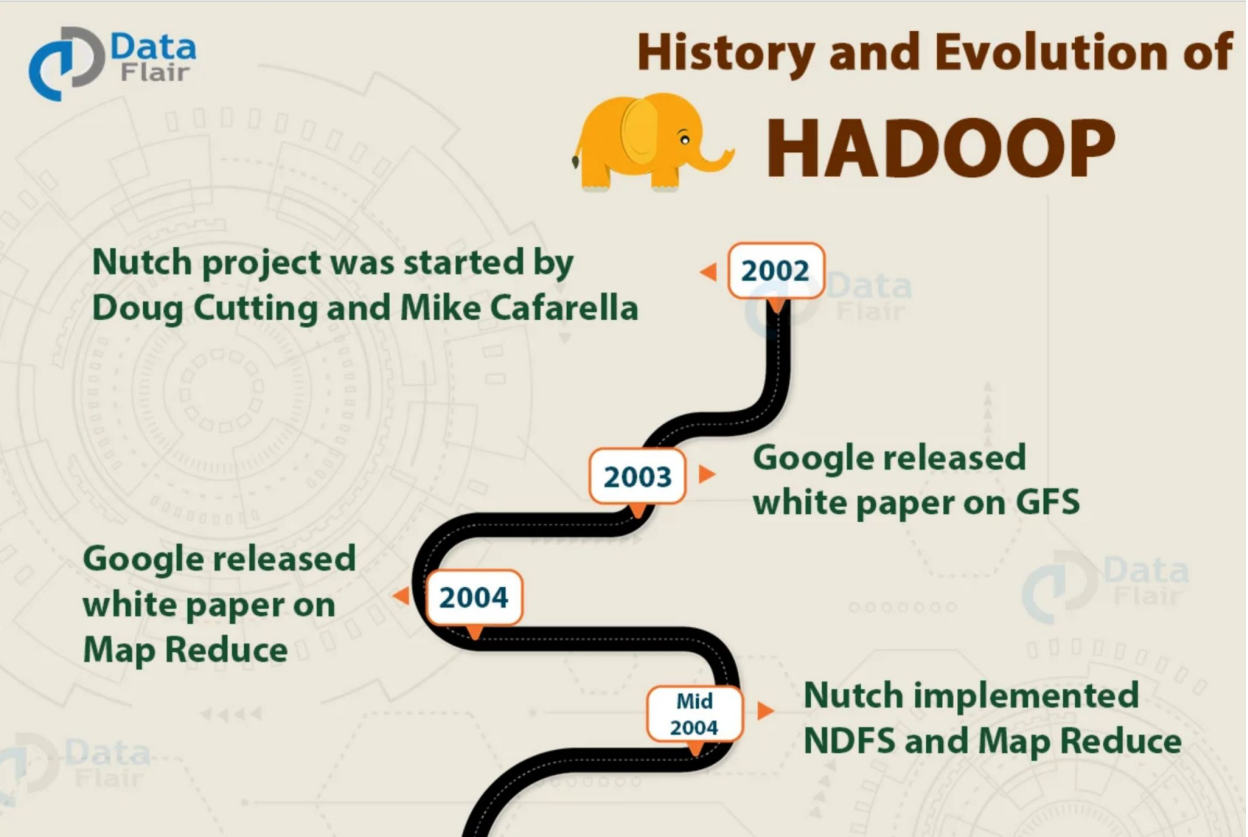
- Hadoop can run Mapreduce programs written in various languages:
  - Java, Ruby, Python, and C++
- Hadoop handles the complexities associated with running on multiple machines (e.g., coordination, reliability, etc.)
- Hadoop MapReduce comes bundled with a library of generally useful mappers, reducers, and partitioners.

MapReduce - Hadoop - Prominent Users

- Amazon
- Adobe
- Alibaba
- AOL
- EBay
- Facebook
- Google
- Hulu
- IBM
- ImageShack
- LinkedIn
- Rackspace
- Rakuten
- Recruit
- Spotify
- Twitter
- Web Alliance
- Yahoo!

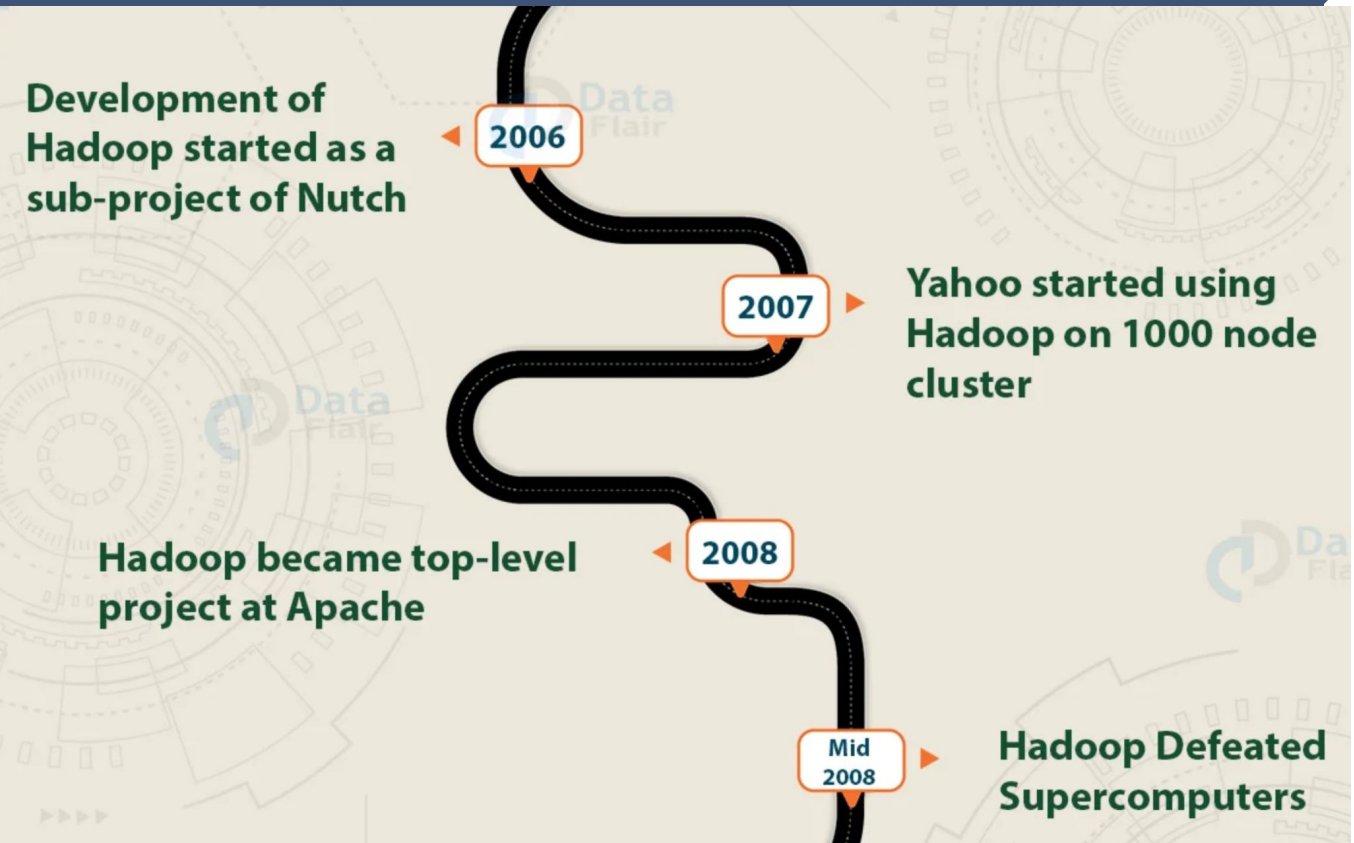
https://cwiki.apache.org/confluence/display/hadoop2/PoweredBy

MapReduce - History of Hadoop

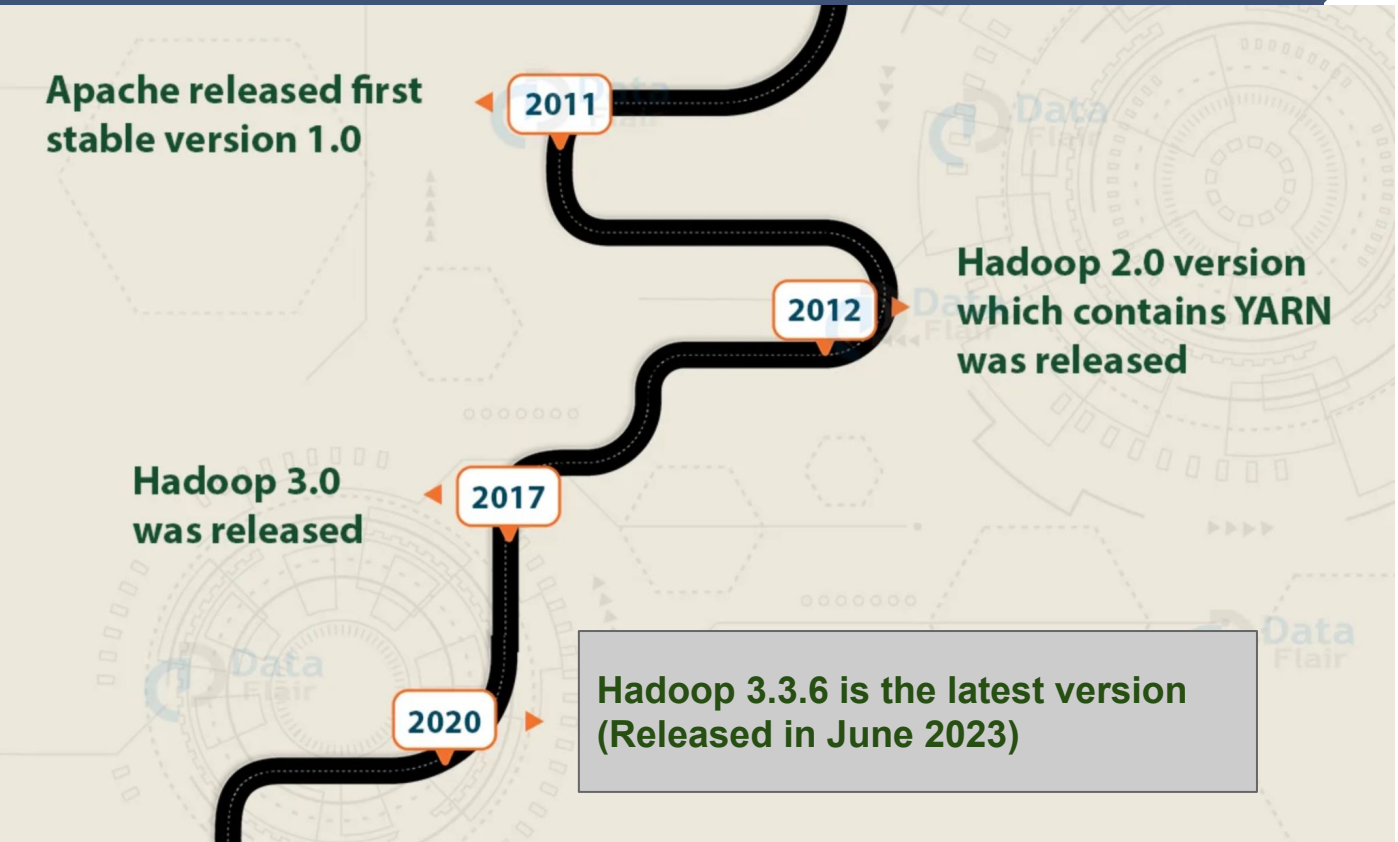


https://data-flair.training/blogs/hadoop-history/

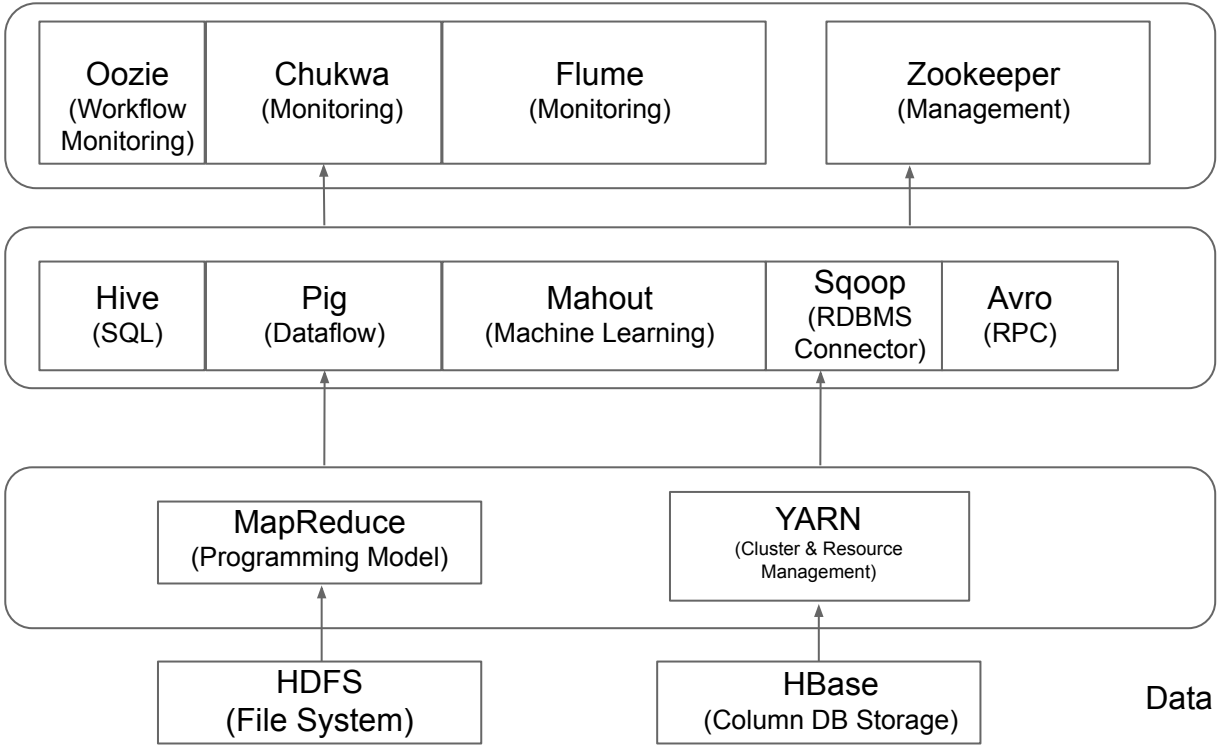
MapReduce - History of Hadoop



MapReduce - History of Hadoop



MapReduce - Hadoop Ecosystem



MapReduce - Hadoop Ecosystem - Data Management

- **Oozie** : a workflow scheduler system to manage Apache Hadoop jobs.
- **Chukwa** : A data collection system for managing large distributed systems
- **Flume** : a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.
- **Zookeeper** : a high-performance coordination service for distributed applications

MapReduce - Hadoop Ecosystem - Data Access

- **Hive** : a data warehouse infrastructure which provides data summarization and ad hoc querying
- **Pig** : a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs
- **Mahout** : a distributed linear algebra framework and mathematically expressive Scala DSL designed to let mathematicians, statisticians, and data scientists quickly implement their own algorithms.
- **Sqoop** : a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.
- **Avro** : a data serialization system

MapReduce - Hadoop Ecosystem - Data Processing

- **MapReduce** : a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.
- **YARN** : splits up the functionalities of resource management and job scheduling/monitoring into separate daemons.

MapReduce - Hadoop Ecosystem - Data Storage

- **HDFS** : a distributed file system that handles large data sets running on commodity hardware.
- **HBase** : a column-oriented non-relational database management system that runs on top of Hadoop Distributed File System (HDFS)

MapReduce - Word count

- Counts the number of occurrences of each word in a given input set
- **Mapper** tokenizes each line from the input file to multiple key value pairs
- **Reducer** combines these key value pairs taking the summation of the occurrence of each key
- **Launching program** defines the MapReduce job and submits the job to the cluster

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

MapReduce - Word count - Launching Program

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

MapReduce - Configuring the jobs - JobConfs

- JobConfs enable controlling jobs
- JobConfs maps from attribute names to string values
- There are set of framework defined attributes which controls how the job is executed
  - conf.set(“mapred.job.name”, “MyApp”);
- In addition applications can add arbitrary values to the JobConf
  - conf.set(“my.integer”, 22);

MapReduce - Launching Program

- Setup and configures
  - The mapper and the reducer to use
  - The output key and value types
  - The locations for input and output
- The launching program submits the job and waits for it to complete

MapReduce - Input and Output Formats

- How input data is read by a MapReduce job is specified by designating an *InputFormat* to be used.
- Similarly how output is written is specified by designating an *OutputFormat*
- *TextInputFormat* and *TextOutputFormat* are the default formats which process line-based text data
- Binary Input
  - **SequenceFileInputFormat**
  - **SequenceFileAsTextInputFormat** - a variant of SequenceFileInputFormat that converts the sequence file's keys and values to Text objects.
  - **SequenceFileAsBinaryInputFormat** - SequenceFileAsBinaryInputFormat is a variant of SequenceFileInputFormat that retrieves the sequence file's keys and values as opaque binary objects.

MapReduce - Input and Output Formats

- Binary Output
  - **SequenceFileOutFormat** - writes sequence files for its output
  - **SequenceFileAsBinaryOutputFormat** - the counterpart to SequenceFileAsBinaryInput Format, and it writes keys and values in raw binary format into a SequenceFile container.
  - **MapFileOutputFormat** - writes MapFiles as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.

MapReduce - Input and Output Formats

- Lazy output
  - Some applications prefer that empty files not be created. This can be achieved via LazyOutputFormat
- Database output
  - The output formats for writing to relational databases and to HBase

MapReduce - Word count - Mapper

```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

MapReduce - Mapper

- Mapper maps input key/value pairs to a set of intermediate key/value pairs.
- These intermediate records do not need to be of the same type as the input records.
- An input pair may map to zero or many output pairs.
- Overall, mapper implementations are passed to the job via *Job.setMapperClass(Class)* method.

MapReduce - Mapper (contd.)

- The framework then calls *map(WritableComparable, Writable, Context)* for each key/value pair in the InputSplit for that task.
- Output pairs do not need to be of the same types as input pairs.
- A given input pair may map to zero or many output pairs.
- Output pairs are collected with calls to *context.write(WritableComparable, Writable)*.

MapReduce - Mapper (contd.)

- All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to the Reducer(s) to determine the final output.
- Users can control the grouping by specifying a Comparator via *Job.setGroupingComparatorClass(Class)*.
- The Mapper outputs are sorted and then partitioned per Reducer.
- The total number of partitions is the same as the number of reduce tasks for the job.
- Users can control which keys (and hence records) go to which Reducer by implementing a custom Partitioner.

MapReduce - How many maps?

- Usually driven by the total size of the inputs.
  - The total number of blocks of the input files
- The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks.
- Task setup takes a while, so it is best if the maps take at least a minute to execute.
- If you expect 10TB of input data and have a block size of 128MB, you'll end up with 82,000 maps, unless *Configuration.set(MRJobConfig.NUM_MAPS, int)* (which only provides a hint to the framework) is used to set it even higher.

MapReduce - How many maps? (Contd.)

- Usually driven by the total size of the inputs.
  - The total number of blocks of the input files (the number of HDFS blocks being processed)
- The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks.
- Task setup takes a while, so it is best if the maps take at least a minute to execute.
- If you expect 10TB of input data and have a block size of 128MB, you'll end up with 82,000 maps, unless *Configuration.set(MRJobConfig.NUM_MAPS, int)* (which only provides a hint to the framework) is used to set it even higher.

MapReduce - How many maps? (Contd.)

- The number of maps can also be controlled by specifying the minimum split size
- The actual sizes of the map inputs are calculated by:
  - max(min(block_size,data/#maps), min_split_size)

MapReduce - Combiner

- Users can optionally specify a combiner, via *Job.setCombinerClass(Class)*, to perform local aggregation of the intermediate outputs.
- This helps to cut down the amount of data transferred from the Mapper to the Reducer (i.e., reduce the size of the transient data).
- The intermediate, sorted outputs are always stored in a simple <key-len, key, value-len, value> format.
- Applications can control if, and how, the intermediate outputs are to be compressed and the *CompressionCodec* to be used via the Configuration.
- Combiners have the same interface as Reduces and often are the same class

MapReduce - Combiner

- Combiners must not introduce side effects, they run an intermediate number of times
- In word count application,

job.setCombinerClass(IntSumReducer.class)

MapReduce - Word count - Reducer

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

MapReduce - Reducer

- Reduces a set of intermediate values which share a key to a smaller set of values.
- Reducer implementation is passed for the job via the *Job.setReducerClass(Class)* method and can override it to initialize themselves
- Framework calls *reduce(WritableComparable, Iterable<Writable>, Context)* method for each *<key, (list of values)>* pair in the grouped inputs

MapReduce - Reducer (Contd.)

- Reducer has three primary phases
  - **Shuffle**
    - Input to the Reducer is the sorted output of the mappers
    - In this phase the framework fetches the relevant partition of the output of all the mappers via HTTP
  - **Sort**
    - In this stage the framework groups Reducer inputs by keys
    - Shuffle and sort phases happen simultaneously
  - **Reduce**
    - the reduce(WritableComparable, Iterable<Writable>, Context) method is called for each <key, (list of values)> pair in the grouped inputs.

MapReduce - Reducer (Contd.)

- The output of the reduce task is typically written to the FileSystem via Context.write(WritableComparable, Writable)
- The output of the Reducer is not sorted

MapReduce - How many reduces?

- Optimal number of reduces is identified to be 0.95 or 1.75 multiplied by (<no. of nodes> * <no. of maximum containers per node>)
- With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish
  - *0.95*num_nodes*mapred.tasktracker.tasks.maximum*
- With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.
  - *1.75*num_nodes*mapred.tasktracker.tasks.maximum*

MapReduce - How many reduces? (Contd.)

- Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.
- The scaling factors above are slightly less than whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.

MapReduce - Reducer NONE

- It is valid to set the number of reduce-tasks to zero if no reduction is required.
- The outputs of the map-tasks go directly to the FileSystem, into the output path set by *FileOutputFormat.setOutputPath(Job, Path)*.
- The framework does not sort/shuffle the map-outputs before writing them out to the FileSystem.

MapReduce - Partitioner

- Partitioner partitions the key space
- Partitioner controls the partitioning of the keys of the intermediate map-outputs
- The key (or a subset of the key) is used to derive the partition, generally by using a hash function.
- Default partitioning spreads keys evenly but randomly
  - Uses key.hashCode() % num_reduces
- Custom partitioning is often needed to produce a total order in the output

MapReduce - Partitioner

- Custom partitioner,
  - Should implement Partitioner interface
  - Set by calling *conf.setPartitionerClass(MyPart.class)*
  - In order to get a total order sample the map output keys and pick values to divide the keys into roughly equal buckets and use that in the partitioner

MapReduce - Counter

- A facility for MapReduce applications to report its statistics.
  - Useful for quality control or for application level-statistics
  - Problem Diagnosing
- Mapper and Reducer implementations can use the Counter to report statistics.
  - E.g., Hadoop counts records in to and out of Mapper and Reducer

MapReduce - Compression

- Huge performance gains can be obtained via compressing the outputs and the intermediate data
  - Can be specified via a configuration file / set programmatically
  - Set *mapred.output.compress* to true to compress job output
  - Set *mapred.compress.map.output* to true to compress map outputs
- Compression Types
  - record - Each value is compressed individually
  - block - Group of keys and values are compressed together
  - Block compression is the most effective approach
- Compression Codecs
  - Default (zlib) - slower, but more compression
  - LZO - faster, but less compression

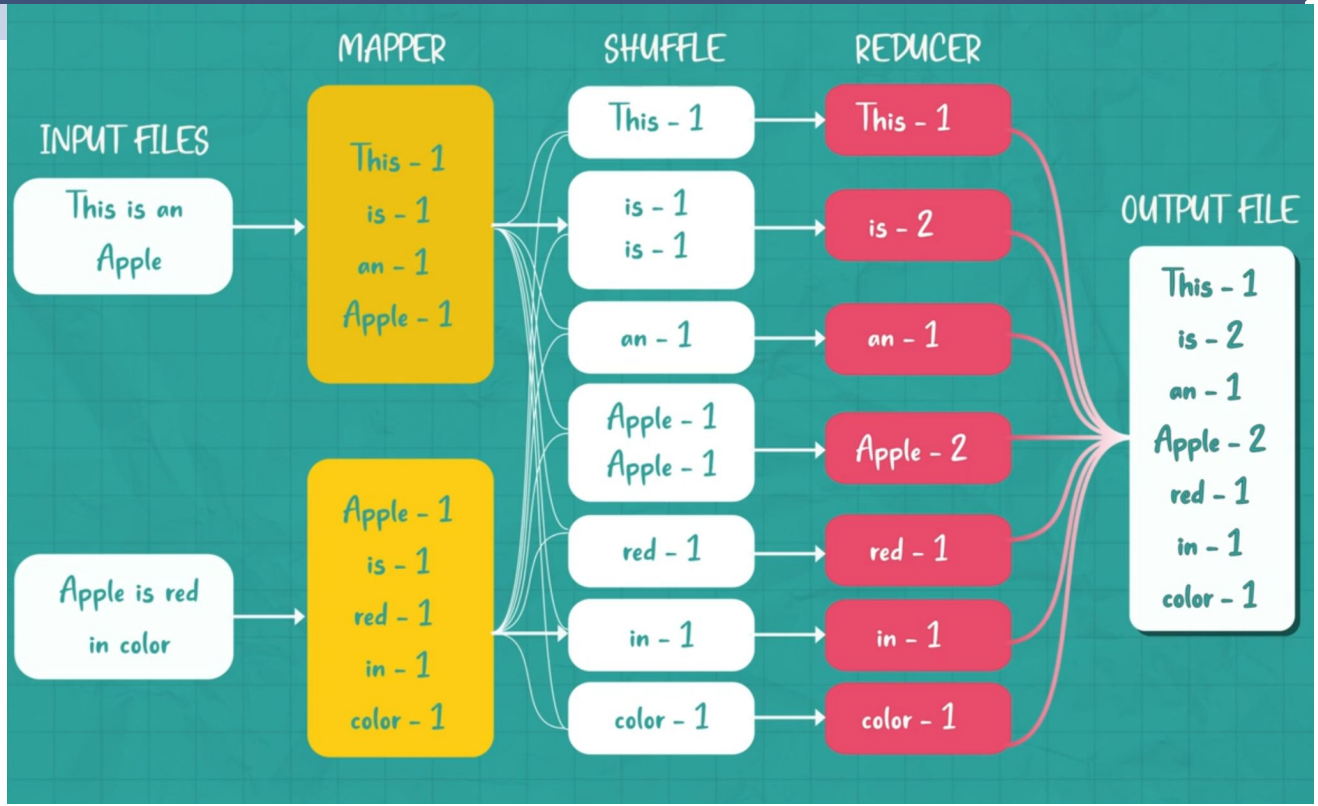
MapReduce - Hadoop - Speculative execution

- Hadoop can run multiple instances of slow tasks
  - the output from the instance that finishes first is used
  - The configuration value *mapred.speculative.execution*
  - Can bring in significant long tails of the jobs

MapReduce - Distributed File Cache

- Some applications require only read only copies of the data on the local computer.
  - Downloading 1GB of data for each Mapper is expensive
- Define a list of items to be downloaded on JobConf.
- Files are downloaded once per computer
- Add to launching program:
- *DistributedCache.addCacheFile(new URI(“hdfs://nn:8020/foo”), conf);*
- Add to task:
- *Path[] files = DistributedCache.getLocalCacheFiles(conf);*

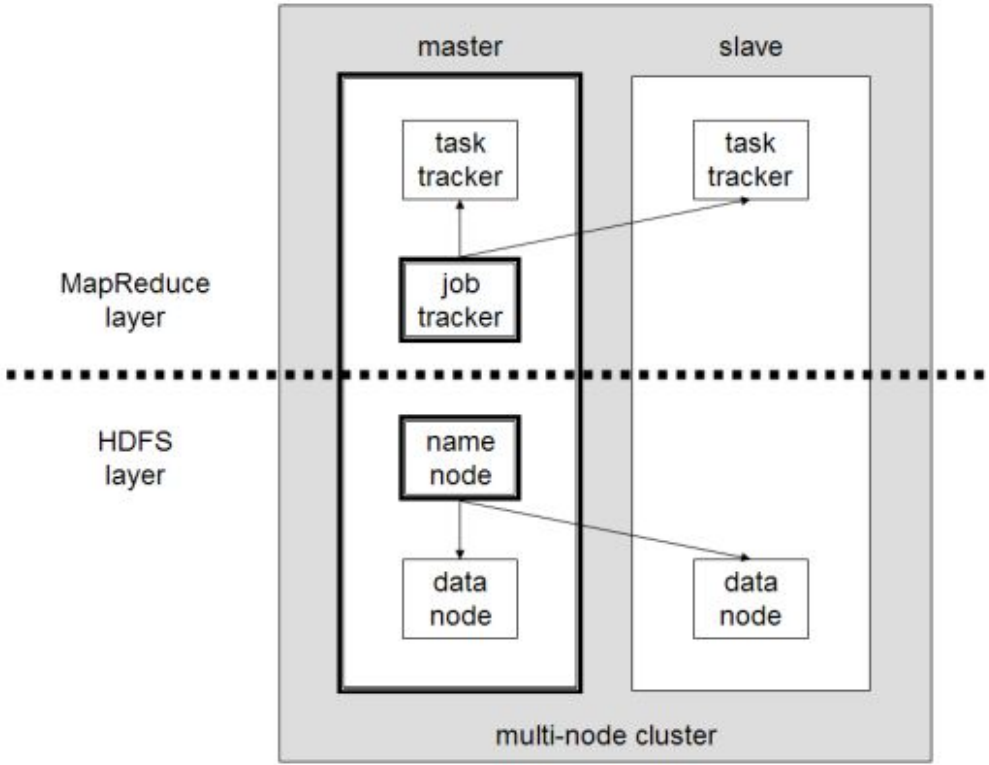
MapReduce



Source : https://www.youtube.com/watch?v=cHGaQz0E7AU

- Word count example

Hadoop



source: https://en.wikipedia.org/wiki/Apache_Hadoop#/media/File:Hadoop_1.png

MapReduce/Hadoop Advantages

- Cheaper way of processing large datasets since it can leverage disk storage of commodity hardware
- Open source
- Fault tolerant (achieved via erasure coding)
- Low network traffic - Hadoop jobs are divided into sub tasks and are assigned to data nodes

https://ieeexplore.ieee.org/document/6680946

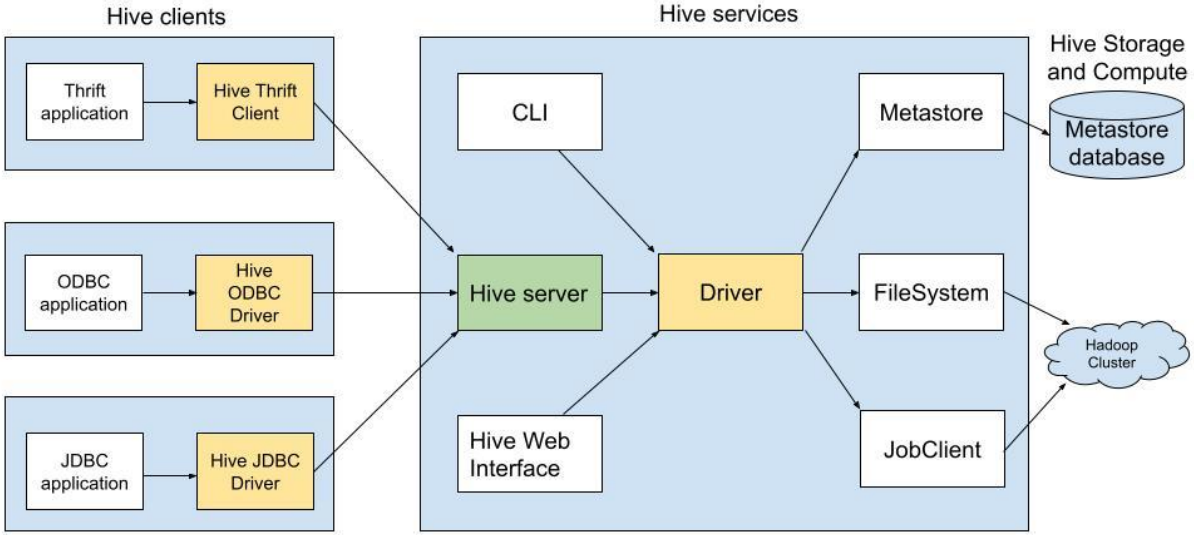
MapReduce/Hadoop Key Limitations

- Require advanced programming skills and deep understanding of the system architecture
- Too "low-level" for analysts used for SQL-like / Declarative languages
- Programming model is batch processing oriented
- Proper performance tuning of Hadoop require knowledge of both available hardware and workload characteristics

https://ieeexplore.ieee.org/document/6680946

Hive

- Developed at Facebook and used heavily for Facebook jobs
- Hive was created to make it possible for analysts with strong SQL skills to run queries on the huge volumes of data that Facebook stored in HDFS



Hive

- "Relational database" built on Hadoop
  - Hive maintains list of table schemas
  - SQL-like query language (HiveQL)
  - Supports table partitioning, complex data types, clustering, some optimizations
  - Can call Hadoop Streaming scripts from HiveQL

Setup Hive Sample

- clone https://github.com/big-data-europe/docker-hive.git
- run *docker-compose up -d*

```
miyurud@miyurud-XPS-15-9510:~/git/docker-hive$ docker-compose up -d
docker-hive_namenode_1 is up-to-date
docker-hive_hive-server_1 is up-to-date
docker-hive_datanode_1 is up-to-date
docker-hive_hive-metastore-postgresql_1 is up-to-date
docker-hive_presto-coordinator_1 is up-to-date
docker-hive_hive-metastore_1 is up-to-date
```

Running Hive Sample

- Run
  docker-compose up -d presto-coordinator

- Run
  docker-compose exec hive-server bash
- Once logged into the hive-server execute Beeline as follows

  /opt/hive/bin/beeline -u jdbc:hive2://localhost:10000

Running Hive Sample

- Hive comes with HiveServer2 which is a server interface and has its own Command Line Interface(CLI) called Beeline.
- Beeline is used to connect to Hive running on Local or Remote server and run HiveQL queries.
- Beeline is a JDBC client that is based on the SQLLine CLI.

Running Hive Sample

- Run
  CREATE TABLE pokes (foo INT, bar STRING);
- Run
  LOAD DATA LOCAL INPATH '/opt/hive/examples/files/kv1.txt' OVERWRITE INTO TABLE pokes;

```
miyurud@miyurud-XPS-15-9510:~/git/docker-hive$ docker-compose exec hive-server bash
root@db80ceb5f599:/opt# CREATE TABLE pokes (foo INT, bar STRING);
bash: syntax error near unexpected token `('
root@db80ceb5f599:/opt# /opt/hive/bin/beeline -u jdbc:hive2://localhost:10000
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/hive/lib/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/hadoop-2.7.4/share/hadoop/common/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Connecting to jdbc:hive2://localhost:10000
Connected to: Apache Hive (version 2.3.2)
Driver: Hive JDBC (version 2.3.2)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 2.3.2 by Apache Hive
0: jdbc:hive2://localhost:10000> CREATE TABLE pokes (foo INT, bar STRING);
No rows affected (0.747 seconds)
0: jdbc:hive2://localhost:10000> LOAD DATA LOCAL INPATH '/opt/hive/examples/files/kv1.txt' OVERWRITE INTO TABLE pokes;
No rows affected (0.877 seconds)
0: jdbc:hive2://localhost:10000>
```

Run query

- Query this setup using PrestoDB
- first download the presto JAR file using the command

```
wget https://repo1.maven.org/maven2/io/prestosql/presto-cli/308/presto-cli-308-executable.jar
```

Make it ready

```
mv presto-cli-308-executable.jar presto.jar
chmod +x presto.jar
```

Run

```
./presto.jar --server localhost:8080 --catalog hive --schema default
```

Run

```
select * from pokes;
```

Run Query

```
presto:default> select * from pokes;
 foo |   bar
-----+---------
 238 | val_238
  86 | val_86
 311 | val_311
  27 | val_27
 165 | val_165
 409 | val_409
 255 | val_255
 278 | val_278
  98 | val_98
 484 | val_484
 265 | val_265
 193 | val_193
 401 | val_401
 150 | val_150
 273 | val_273
 224 | val_224
```

```
presto:default> select COUNT(*) from pokes;
 _col0
-------
   500
(1 row)

Query 20231007_181044_00007_qf643, FINISHED, 1 node
Splits: 18 total, 18 done (100.00%)
0:00 [500 rows, 5.68KB] [4.6K rows/s, 52.3KB/s]

presto:default>
```

Spark

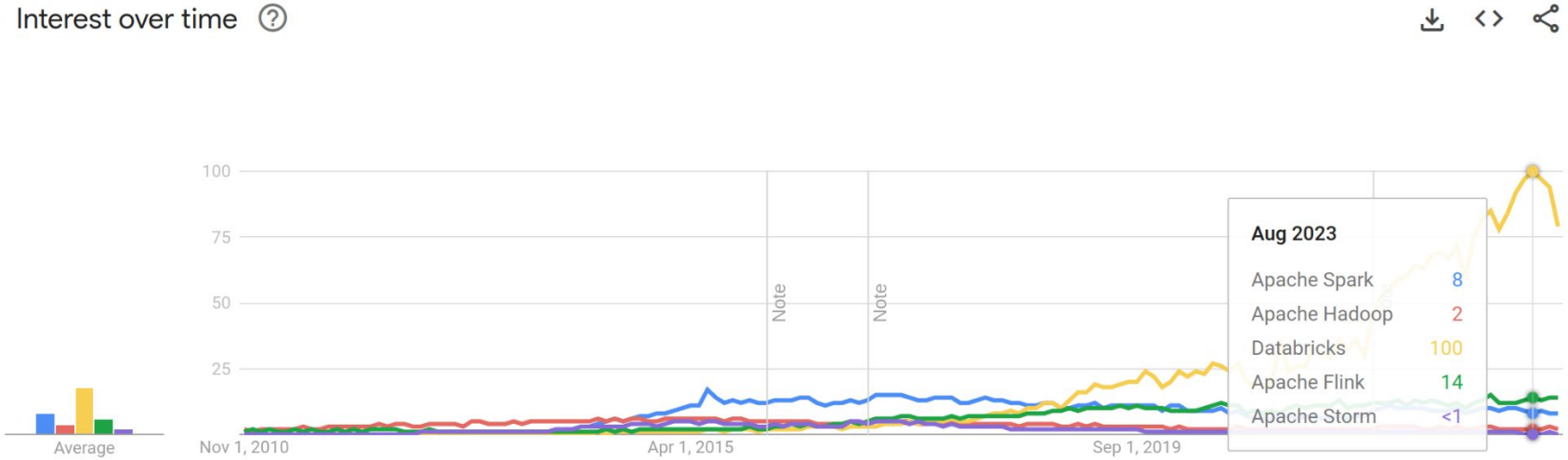
- Spark is a generalized framework for distributed data processing providing functional API for manipulating data at scale, in-memory data caching, and reuse across computations.

- The goal of Spark was to provide a fast general purpose cluster framework for large scale distributed data processing.

History of Spark

- Spark project was initiated by a group of researchers from UC Berkeley.
- It was given to Apache software foundation in 2013.
- Currently, Apache Spark is one of the most widely-used engines for scalable computing.
- Thousands of companies, including 80% of the Fortune 500, use Apache Spark

Hadoop, Spark, and Databricks on Google trends



Databricks

- Databricks was founded in 2013 by the original creators of Apache Spark™, Delta Lake and MLflow
- Today, more than 9,000 organizations worldwide rely on Databricks to enable massive-scale data engineering, collaborative data science, full-lifecycle machine learning and business analytics.

Spark

- Spark does in-memory data caching and reuse across computations.
- Spark applies a set of coarse-grained transformations over partitioned data
- Spark relies on the dataset lineage to recompute tasks in case of failures.
- Spark supports multiple input formats and integrates with resource management and scheduling frameworks such as Apache Mesos / YARN

Spark Pillars

- Two main abstractions of Spark
  - RDD - Resilient Distributed Dataset
  - DAG - Directed Acyclic Graph

Spark - Resilient Distributed Dataset

- RDD is the primary data abstraction in Apache Spark and the core of Spark
- A programming abstraction that represents a collection of read only objects (i.e., records) split across a computing cluster.
- RDDs are computed over many JVMs (however, Scala collections live on single JVM)
- RDDs are fault-tolerant, immutable, and parallel data structure

Spark - Resilient Distributed Dataset

- RDDs are a container of instructions on how to materialize big data and how to split the data into manageable partitions.
- An RDD belong to one and only one SparkContext.

Spark - Resilient Distributed Dataset

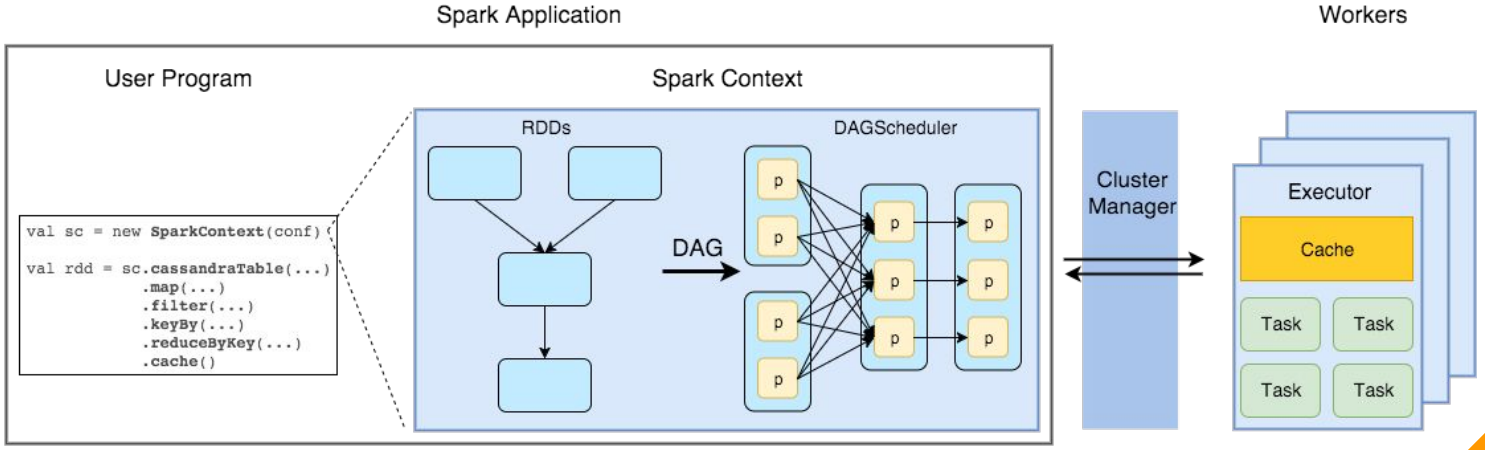
- RDDs can be created through deterministic operation
  - using text files, SQL databases, No-SQL databases, distributed file systems (e.g., HDFS), Cloud Storage
  - from another RDD
- Stores information about parent RDDs
  - used for execution optimization and operations pipelining
  - to recompute the data in case of failure
- RDDs allow standard Mapreduce functions, joining datasets, filtering, and aggregation

Spark - Resilient Distributed Dataset

- Processing of RDDs is done entirely in memory
  - Offers tremendous performance improvements (e.g., 100x faster compared to MapReduce)
- RDDs are designed to hide complexities from users
- Provides API for
  - Manipulating the collection of elements
  - Persisting intermediate results in memory for later reuse
  - Controlling partitioning to optimize data placement

Spark Application

- Spark Application (Also referred to as Driver Program or Application Master) at high-level consists of SparkContext and user program which interacts with it creating RDDs and performing series of transformations to achieve the final result.



https://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark

Spark - RDD from a developer perspective

- RDD represents distributed immutable data structure (partitioned data + iterator) and lazily evaluated operations (transformations)

```
1   // a list of partitions (e.g., splits in Hadoop)
2   def getPartitions: Array[Partition]
3
4   // a list of dependencies on other RDDs
5   def getDependencies: Seq[Dependency[_]]
6
7   // a function for computing each split
8   def compute(split: Partition, context: TaskContext) : Iterator[T]
9
10  // (optional) a list of preferred locations to compute each split on
11  def getPreferredLocations(split: Partition) : Seq[String] = Nil
12
13  // (optional) a partitioner for key-value RDDs
14  val partitioner: Option[Partitioner] = None
```

Lineage

Execution optimization

Example of RDD Creation

- sparkContext.textFile("hdfs://...")

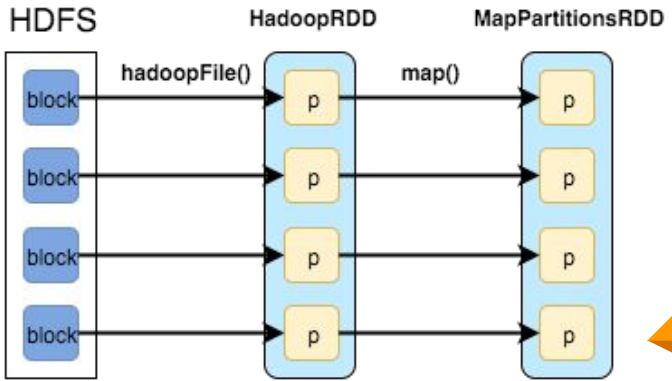
HadoopRDD:

- getPartitions = HDFS blocks
- getDependencies = None
- compute = load block in memory
- getPrefferedLocations = HDFS block locations
- partitioner = None

First loads HDFS blocks in memory and then applies map() to filter out keys creating two RDDs

MapPartitionsRDD

- getPartitions = same as parent
- getDependencies = parent RDD
- compute = compute parent and apply map()
- getPrefferedLocations = same as parent
- partitioner = None



RDD Operations

- Operations on RDDs can be divided into several groups
- **Transformations**
  - apply aggregation function to the whole dataset (groupBy, sortBy)
  - apply user function to every element in a partition (or to the whole partition)
  - provide functionality for repartitioning (repartition, partitionBy)
  - introduce dependencies between RDDs to form DAG

RDD Operations

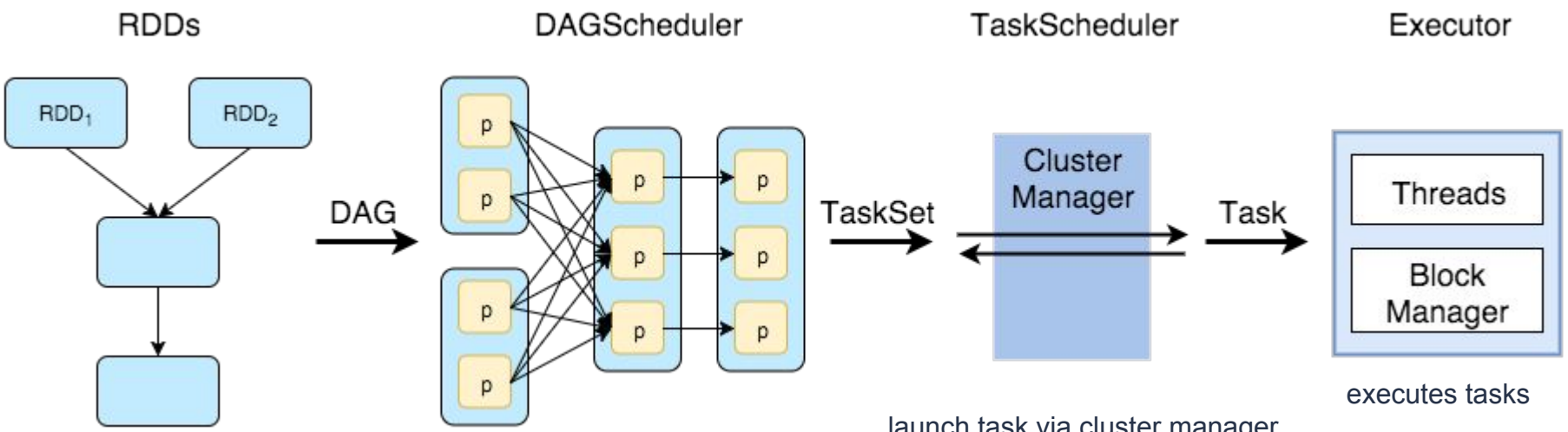
- **Actions**
  - trigger job execution
  - used to materialize computation results
- **Extra: persistence**
  - explicitly store RDDs in memory, on disk, or off-heap (cache, persist)
  - checkpointing for truncating RDD lineage

Code sample : joining aggregated and raw data

- A Spark job which aggregates data from Cassandra in lambda style merging previously aggregated data from raw storage and demonstrates some of the transformations and actions available on RDDs.

```
//aggregate events after specific date for given campaign
val events =
    sc.cassandraTable("demo", "event")
      .map(_.toEvent)
      .filter { e =>
        e.campaignId == campaignId && e.time.isAfter(watermark)
      }
      .keyBy(_.eventType)
      .reduceByKey(_ + _)
      .cache()

//aggregate campaigns by type
val campaigns =
    sc.cassandraTable("demo", "campaign")
      .map(_.toCampaign)
      .filter { c =>
        c.id == campaignId && c.time.isBefore(watermark)
      }
      .keyBy(_.eventType)
      .reduceByKey(_ + _)
      .cache()

//joined rollups and raw events
val joinedTotals = campaigns.join(events)
            .map { case (key, (campaign, event)) =>
              CampaignTotals(campaign, event)
             }
            .collect()

//count totals separately
val eventTotals =
    events.map{ case (t, e) => s"$t -> ${e.value}" }
    .collect()

val campaignTotals =
    campaigns.map{ case (t, e) => s"$t -> ${e.value}" }
    .collect()
```

Execution Workflow



rdd1.join(rdd2)
.groupBy(...).filter(...)

split graph into stages of tasks

submit each stage as ready

launch task via cluster manager

retries struggling or failed tasks

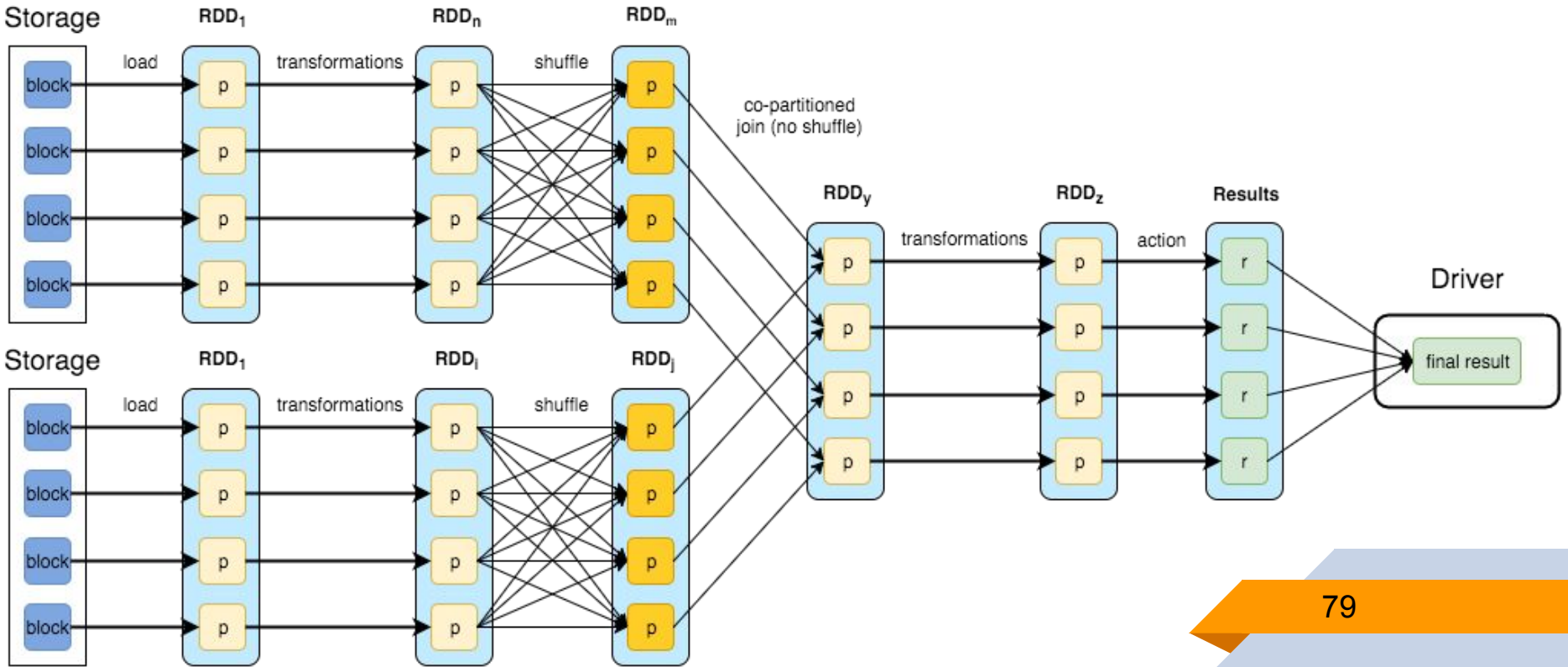
executes tasks

stores and serves blocks

Execution Workflow

- User code containing RDD transformations form Direct Acyclic Graph
- It is then split into stages of tasks by DAGScheduler
- Stages combine tasks which do not require shuffling/repartitioning of the data
- Tasks are run on workers and results are returned to the client

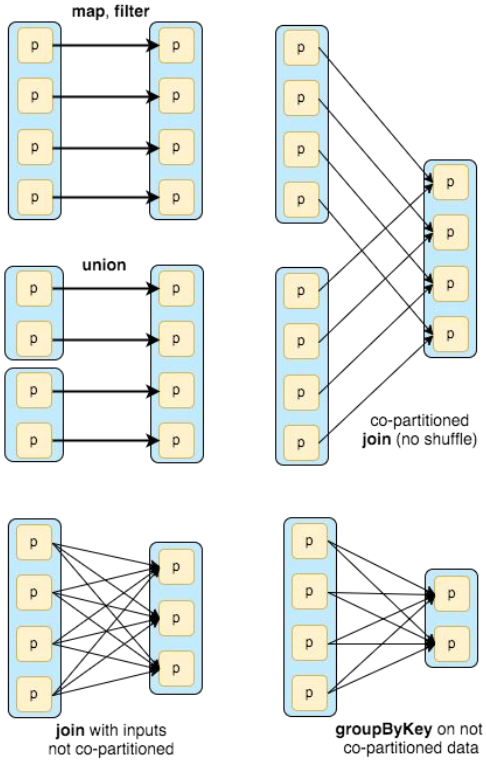
DAG for the code sample



DAG for the code sample (Contd.)

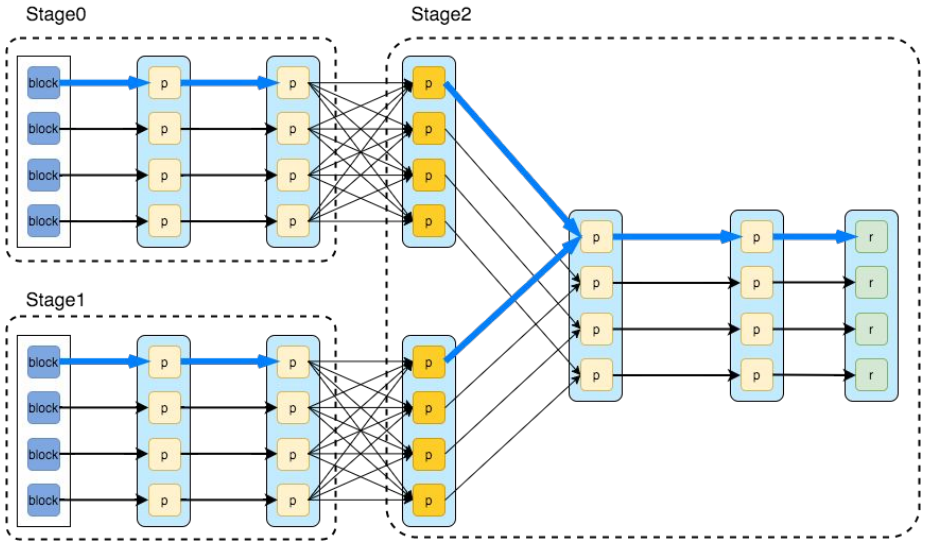
- Data processing workflow
  - reading the data source
  - applying a set of transformations
  - materializing the results in various forms
- Transformations create dependencies between RDDs

Dependency Types



- **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 dividing the RDD graph at shuffle boundaries
- RDD operations with narrow dependencies (e.g., map() and filter()) are pipelined together into one set of tasks in each stage operations with shuffle dependencies require multiple stages
- Eventually 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.

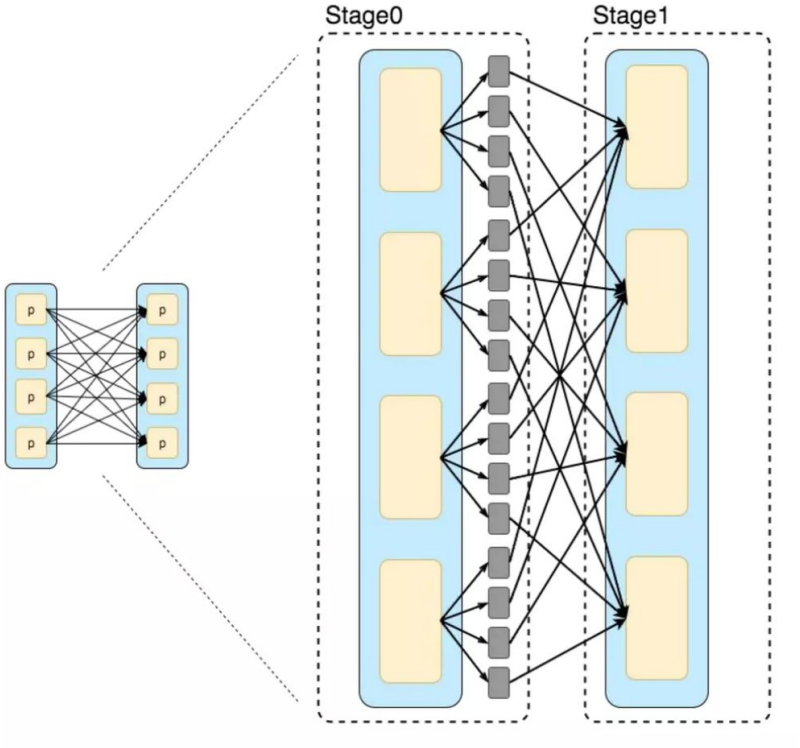
Spark Tasks

- Two types of tasks in Spark
- **ShuffleMapTask**
  - Partitions its input for shuffle. writes the result of executing a serialized task code over the records (of a RDD partition) to the shuffle system and returns a MapStatus
- **ResultTask**
  - Sends output to the driver

Spark Stages

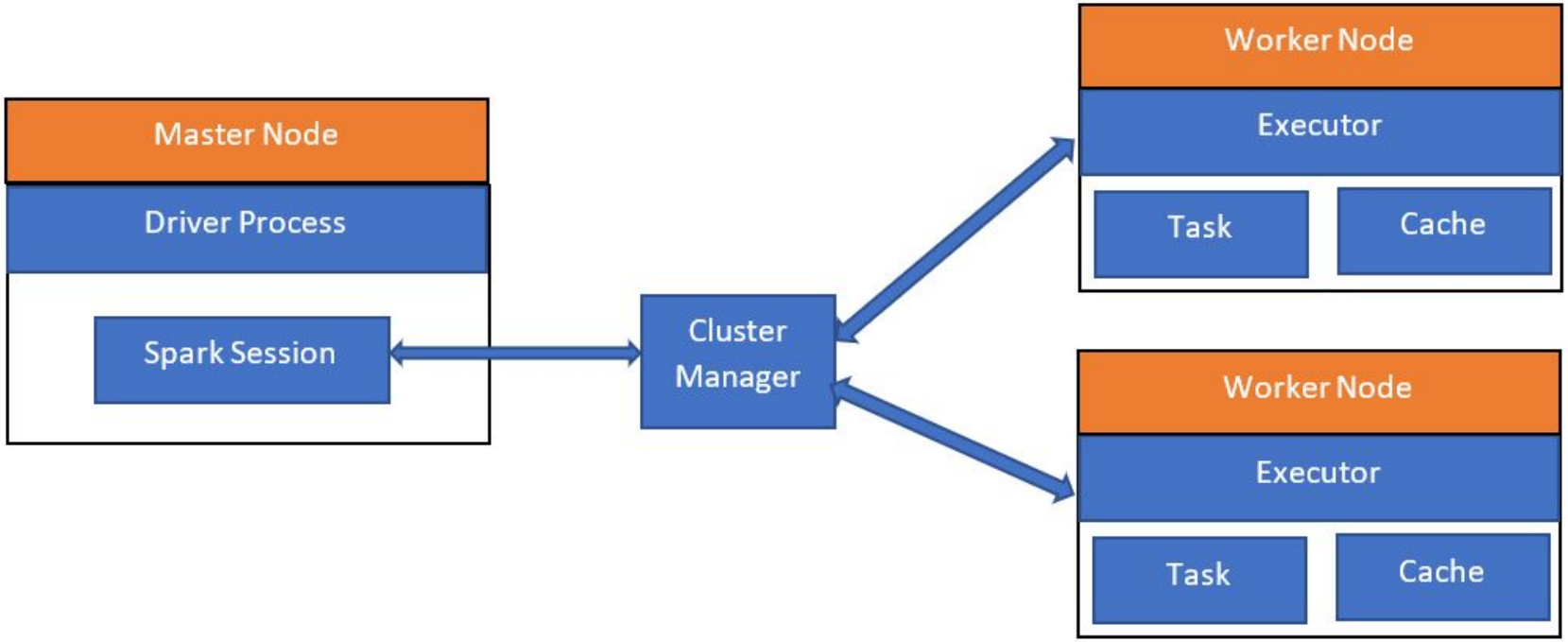
- Two types of stages in Spark
  - **ShuffleMapStage**
    - an intermediate stage in the execution DAG that produces data for shuffle operation. It is an input for the other following stages in the DAG of stages.
  - **ResultStage**
    - is the final stage in a job that applies a function to one or many partitions of the target RDD to compute the result of an action.

Shuffle



- Shuffle write
  - redistributes data among partitions and writes files to disk
  - each hash shuffle task creates one file per “reduce” task (total = MxR)
  - sort shuffle task creates one file with regions assigned to the reducer
  - sort shuffle uses in-memory sorting with spillover to disk to get the final result
- Shuffle read
  - fetches the files and applies reduce() logic
  - if data ordering is needed then it is sorted on the “reducer” side for any type of shuffle

Spark



source : https://blog.knoldus.com/basic-things-related-to-apache-spark/

Spark - Drivers and Executors

- Processing of RDDs is done via the drivers and executors
- When a program executes it starts via a driver which creates a Spark context (Orchestrator).
- Generates physical plan and uses the cluster manager to coordinate all the executors to schedule and run the tasks.
- The scheduler within a Spark context is called a DAG which assigns the tasks to the worker nodes

Spark - Drivers and Executors (Contd.)

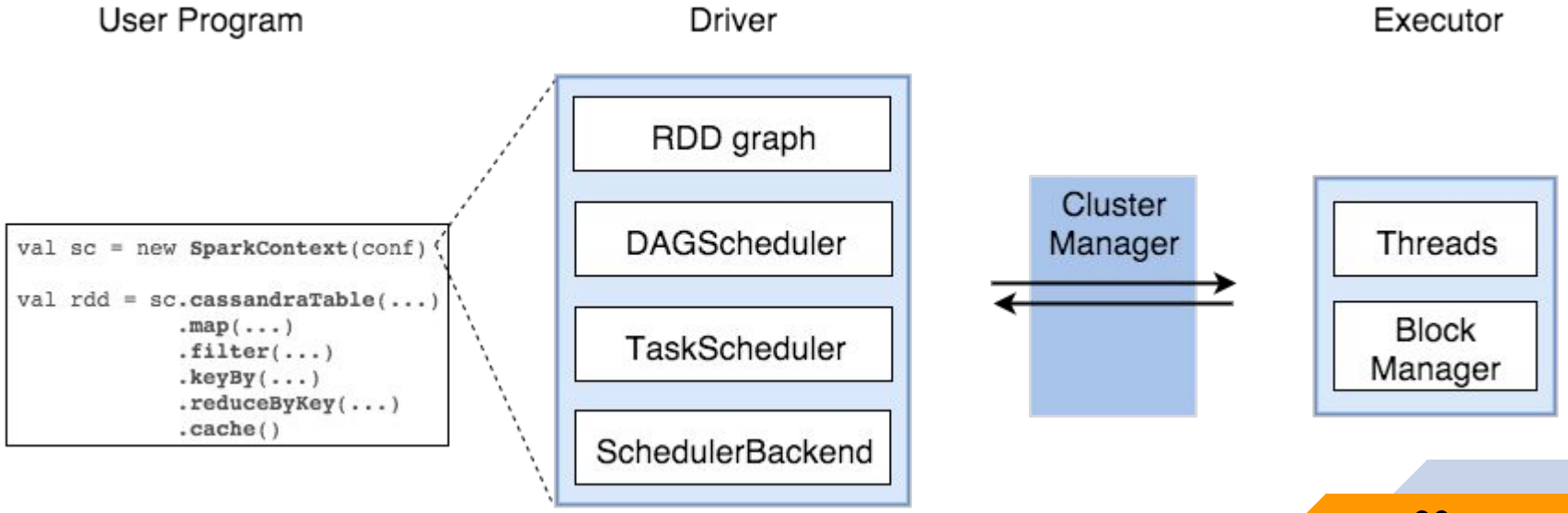
- Executors run tasks scheduled by the driver
- Executors store computation results in memory, on disk or off-heap
- Executors interact with storage systems

Spark - Cluster Manager

- Mesos
- YARN
- Spark Standalone

Spark Driver

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



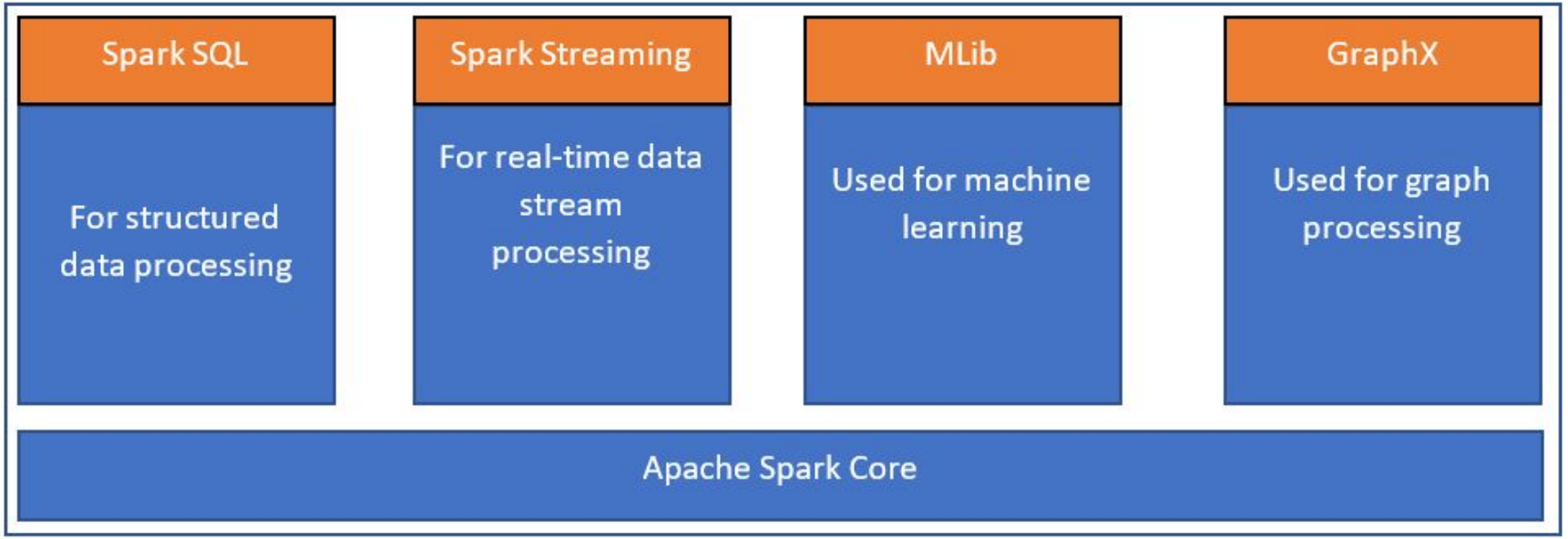
Spark Driver (Contd.)

- 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

Spark Driver (Contd.)

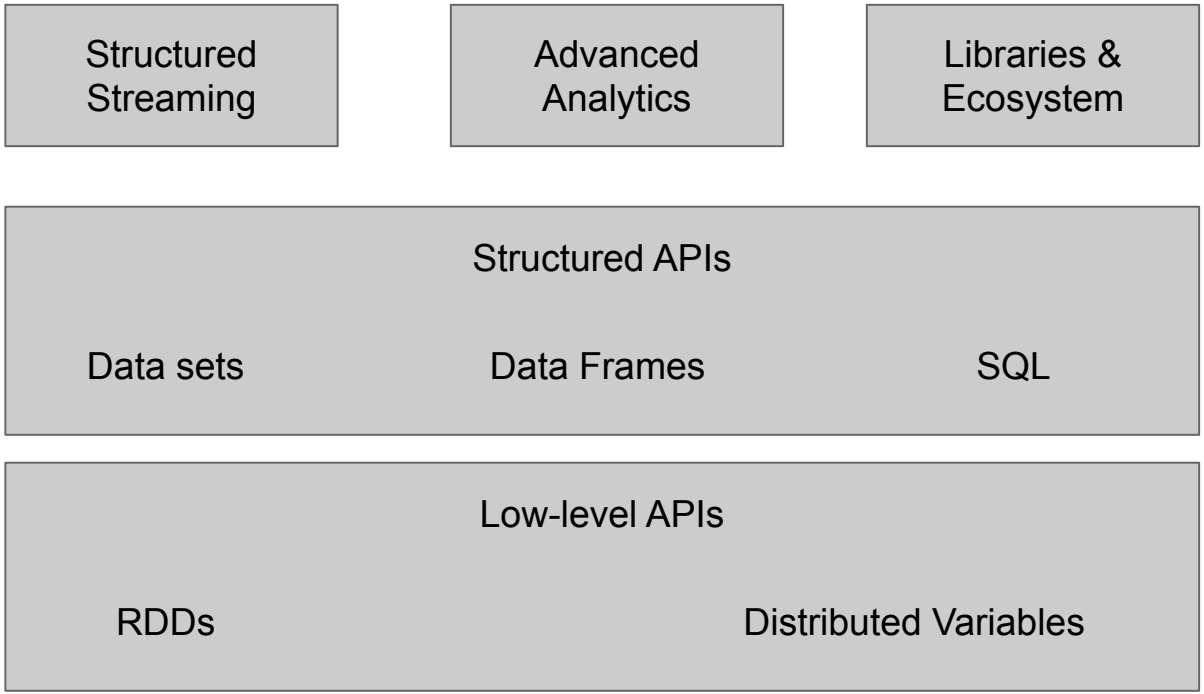
- TaskScheduler
  - responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers
- SchedulerBackend
  - backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)
- BlockManager
  - provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

Spark Stack



source : https://sqlrelease.com/big-data-processing-using-apache-spark-introduction

Spark's Toolkit



Spark word count

```
1   import sys
2   from operator import add
3
4   from pyspark.sql import SparkSession
5
6
7   if __name__ == "__main__":
8       if len(sys.argv) != 2:
9           print("Usage: wordcount <file>", file=sys.stderr)
10          sys.exit(-1)
11
12      spark = SparkSession\
13          .builder\
14          .appName("PythonWordCount")\
15          .getOrCreate()
16
17      lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
18      counts = lines.flatMap(lambda x: x.split(' ')) \
19                    .map(lambda x: (x, 1)) \
20                    .reduceByKey(add)
21      output = counts.collect()
22      for (word, count) in output:
23          print("%s: %i" % (word, count))
24
25      spark.stop()
```

Spark - Advantages

- Speed and performance better than Hadoop (x100)
- Developer friendly tools. Offers popular language bindings for Python, R, Java, and Scala
- Libraries for machine learning and graph analysis

Spark - Main Limitations

- Does not come with its own file management system
- Does not support complete stream processing
  - Events are divided into small batches and each batch is handled as Spark RDD
- Spark memory consumption is very high -> Cost of Spark is very high
- Automatic Code Optimization process is absent in Apache Spark.

Conclusion

- Evolution of Big Data Processing Technologies
- MapReduce
- Hadoop
- Spark

Thank you!