Programming for Big Data

Large Scale Data Processing

Saeed Iqbal Khattak

Centre for Healthcare Modelling & Informatics
Faculty of Information Technology,
University of Central Punjab, Lahore

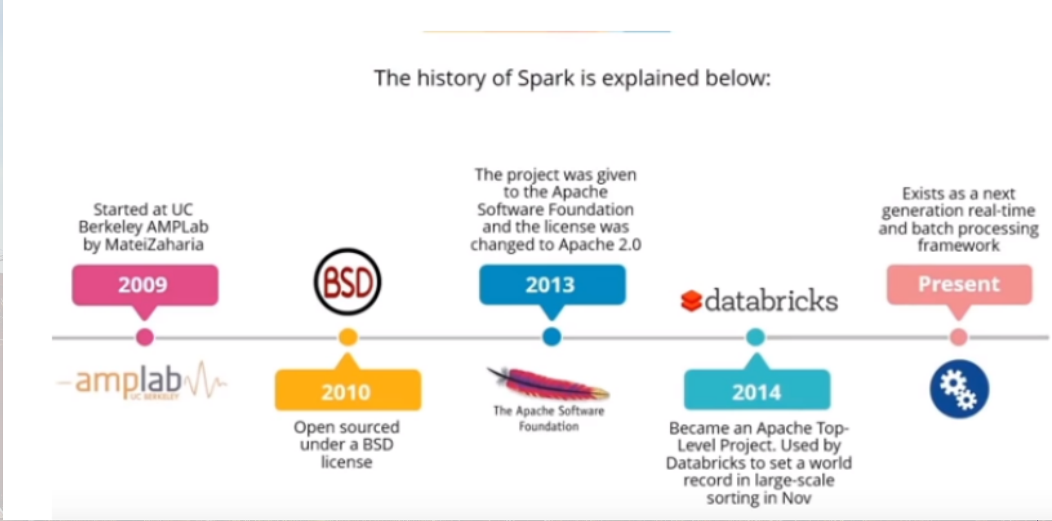
June 8, 2021

Outline

- History of Spark
- What is Spark
- Hadoop vs Spark
- Components of Apache Spark
- Spark Architecture
- Application of Spark
- Spark Use Cases

Assignment 4

[**Assignment 4** – Download Hadoop `hadoop.apache.org/`, install, Configure and run test program (word count)]



- Spark was first open sourced in March 2010, and was transferred to the Apache Software Foundation in June 2013, where it is now a top-level project.

What is Spark

- Apache Spark is a general-purpose and lightning fast cluster computing system:

What is Spark

- Apache Spark is a general-purpose and lightning fast cluster computing system:
- An open-source, distributed processing system used for big data workloads.

What is Spark

- Apache Spark is a general-purpose and lightning fast cluster computing system:
- An open-source, distributed processing system used for big data workloads.
- Utilizes in-memory caching

What is Spark

- Apache Spark is a general-purpose and lightning fast cluster computing system:
- An open-source, distributed processing system used for big data workloads.
- Utilizes in-memory caching
- Optimized query execution for fast analytic queries against data of any size.

What is Spark

- Apache Spark is a general-purpose and lightning fast cluster computing system:
- An open-source, distributed processing system used for big data workloads.
- Utilizes in-memory caching
- Optimized query execution for fast analytic queries against data of any size.
- Provides development APIs in Java, Scala, Python and R.

What is Spark

- Apache Spark is a general-purpose and lightning fast cluster computing system:
- An open-source, distributed processing system used for big data workloads.
- Utilizes in-memory caching
- Optimized query execution for fast analytic queries against data of any size.
- Provides development APIs in Java, Scala, Python and R.
- Internet powerhouses such as Netflix, Yelp, Zillow, and eBay have deployed Spark at massive scale, collectively processing multiple petabytes of data on clusters of over 8,000 nodes.

Why Spark Matters

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Spark: It is also a part of Hadoop Ecosystem. It is a batch processing System at heart too but it also supports stream processing.

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Spark: It is also a part of Hadoop Ecosystem. It is a batch processing System at heart too but it also supports stream processing.

2. **Streaming Engine**

Hadoop: It takes large data set in the input, all at once, processes it.

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Spark: It is also a part of Hadoop Ecosystem. It is a batch processing System at heart too but it also supports stream processing.

2. **Streaming Engine**

Hadoop: It takes large data set in the input, all at once, processes it.

Spark: It processes data streams in micro-batches.

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Spark: It is also a part of Hadoop Ecosystem. It is a batch processing System at heart too but it also supports stream processing.

2. **Streaming Engine**

Hadoop: It takes large data set in the input, all at once, processes it.

Spark: It processes data streams in micro-batches.

3. **Data Flow**

Hadoop: It is a chain of stages. At each stage, you progress forward using an output of the previous stage and producing input for the next stage.

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Spark: It is also a part of Hadoop Ecosystem. It is a batch processing System at heart too but it also supports stream processing.

2. **Streaming Engine**

Hadoop: It takes large data set in the input, all at once, processes it.

Spark: It processes data streams in micro-batches.

3. **Data Flow**

Hadoop: It is a chain of stages. At each stage, you progress forward using an output of the previous stage and producing input for the next stage.

Spark: It represents it as (DAG) Direct Acyclic Graph.

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Spark: It is also a part of Hadoop Ecosystem. It is a batch processing System at heart too but it also supports stream processing.

2. **Streaming Engine**

Hadoop: It takes large data set in the input, all at once, processes it.

Spark: It processes data streams in micro-batches.

3. **Data Flow**

Hadoop: It is a chain of stages. At each stage, you progress forward using an output of the previous stage and producing input for the next stage.

Spark: It represents it as (DAG) Direct Acyclic Graph.

4. **Computation Model**

Hadoop: It has a batch-oriented model. Batch is processing data at rest. It takes a large amount of data at once

Apache Hadoop vs Spark

1. **Data Processing**

Hadoop: It built for batch processing. It takes large data set in the input, all at once, processes it and produces the result.

Spark: It is also a part of Hadoop Ecosystem. It is a batch processing System at heart too but it also supports stream processing.

2. **Streaming Engine**

Hadoop: It takes large data set in the input, all at once, processes it.

Spark: It processes data streams in micro-batches.

3. **Data Flow**

Hadoop: It is a chain of stages. At each stage, you progress forward using an output of the previous stage and producing input for the next stage.

Spark: It represents it as (DAG) Direct Acyclic Graph.

4. **Computation Model**

Hadoop: It has a batch-oriented model. Batch is processing data at rest. It takes a large amount of data at once

Spark: It has adopted micro-batching. Micro-batches are an essentially "collect and then process".

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Spark: It can be 100_x faster than Hadoop for large scale data processing.

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Spark: It can be 100$_x$ faster than Hadoop for large scale data processing.

6. **Memory management**

Hadoop: It provides configurable Memory management. You can do it dynamically or statically.

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Spark: It can be 100_x faster than Hadoop for large scale data processing.

6. **Memory management**

Hadoop: It provides configurable Memory management. You can do it dynamically or statically.

Spark: It provides configurable memory management. After Spark 1.6 versions has moved towards automating memory management.

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Spark: It can be 100_x faster than Hadoop for large scale data processing.

6. **Memory management**

Hadoop: It provides configurable Memory management. You can do it dynamically or statically.

Spark: It provides configurable memory management. After Spark 1.6 versions has moved towards automating memory management.

7. **Fault tolerance**

Hadoop: MapReduce is highly fault-tolerant.

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Spark: It can be 100_x faster than Hadoop for large scale data processing.

6. **Memory management**

Hadoop: It provides configurable Memory management. You can do it dynamically or statically.

Spark: It provides configurable memory management. After Spark 1.6 versions has moved towards automating memory management.

7. **Fault tolerance**

Hadoop: MapReduce is highly fault-tolerant.

Spark: It recovers lost work and with no extra code or configuration.

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Spark: It can be 100_x faster than Hadoop for large scale data processing.

6. **Memory management**

Hadoop: It provides configurable Memory management. You can do it dynamically or statically.

Spark: It provides configurable memory management. After Spark 1.6 versions has moved towards automating memory management.

7. **Fault tolerance**

Hadoop: MapReduce is highly fault-tolerant.

Spark: It recovers lost work and with no extra code or configuration.

8. **Scalability**

Hadoop: MapReduce has incredible scalability potential and has been used in production on tens of thousands of Nodes.

Apache Hadoop vs Spark

5. **Performance**

Hadoop: It doesn't process streamed data hence performance is slower when compared.

Spark: It can be 100_x faster than Hadoop for large scale data processing.

6. **Memory management**

Hadoop: It provides configurable Memory management. You can do it dynamically or statically.

Spark: It provides configurable memory management. After Spark 1.6 versions has moved towards automating memory management.

7. **Fault tolerance**

Hadoop: MapReduce is highly fault-tolerant.

Spark: It recovers lost work and with no extra code or configuration.

8. **Scalability**

Hadoop: MapReduce has incredible scalability potential and has been used in production on tens of thousands of Nodes.

Spark: It is highly scalable, we can keep adding n number of nodes in the cluster. A large known sSpark cluster is of 8000 nodes.

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Spark: It iterates its data in batches. In Spark, each iteration has to be scheduled and executed separately.

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Spark: It iterates its data in batches. In Spark, each iteration has to be scheduled and executed separately.

10. **Language Support**

Hadoop: It Supports Primarily Java, other languages supported are C and C++

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Spark: It iterates its data in batches. In Spark, each iteration has to be scheduled and executed separately.

10. **Language Support**

Hadoop: It Supports Primarily Java, other languages supported are C and C++

Spark: It supports Java, Scala, Python and R. Spark is implemented in Scala. It provides API in other languages like Java, Python, and R.

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Spark: It iterates its data in batches. In Spark, each iteration has to be scheduled and executed separately.

10. **Language Support**

Hadoop: It Supports Primarily Java, other languages supported are C and C++

Spark: It supports Java, Scala, Python and R. Spark is implemented in Scala. It provides API in other languages like Java, Python, and R.

11. **Optimization**

Hadoop: MapReduce, jobs have to be manually optimized, e.g. Configure your cluster correctly, use a combiner, use LZO compression, tune the number of MapReduce Task.

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Spark: It iterates its data in batches. In Spark, each iteration has to be scheduled and executed separately.

10. **Language Support**

Hadoop: It Supports Primarily Java, other languages supported are C and C++

Spark: It supports Java, Scala, Python and R. Spark is implemented in Scala. It provides API in other languages like Java, Python, and R.

11. **Optimization**

Hadoop: MapReduce, jobs have to be manually optimized, e.g. Configure your cluster correctly, use a combiner, use LZO compression, tune the number of MapReduce Task.

Spark: Jobs have to be manually optimized. There is a new extensible optimizer, **Catalyst**.

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Spark: It iterates its data in batches. In Spark, each iteration has to be scheduled and executed separately.

10. **Language Support**

Hadoop: It Supports Primarily Java, other languages supported are C and C++

Spark: It supports Java, Scala, Python and R. Spark is implemented in Scala. It provides API in other languages like Java, Python, and R.

11. **Optimization**

Hadoop: MapReduce, jobs have to be manually optimized, e.g. Configure your cluster correctly, use a combiner, use LZO compression, tune the number of MapReduce Task.

Spark: Jobs have to be manually optimized. There is a new extensible optimizer, **Catalyst**.

12. **Security**

Hadoop: It supports Kerberos authentication, which is somewhat painful to manage.

Apache Hadoop vs Spark

9. **Iterative Processing**

Hadoop: It does not support iterative processing.

Spark: It iterates its data in batches. In Spark, each iteration has to be scheduled and executed separately.

10. **Language Support**

Hadoop: It Supports Primarily Java, other languages supported are C and C++

Spark: It supports Java, Scala, Python and R. Spark is implemented in Scala. It provides API in other languages like Java, Python, and R.

11. **Optimization**

Hadoop: MapReduce, jobs have to be manually optimized, e.g. Configure your cluster correctly, use a combiner, use LZO compression, tune the number of MapReduce Task.

Spark: Jobs have to be manually optimized. There is a new extensible optimizer, **Catalyst**.

12. **Security**

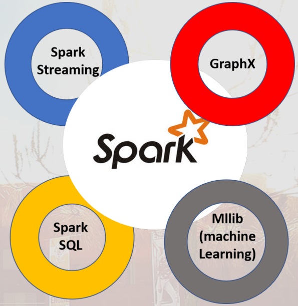
Hadoop: It supports Kerberos authentication, which is somewhat painful to manage.

Spark: The security bonus that Spark can enjoy is that if you run Spark on HDFS.

Components of Spark

There are five main components of Apache Spark:

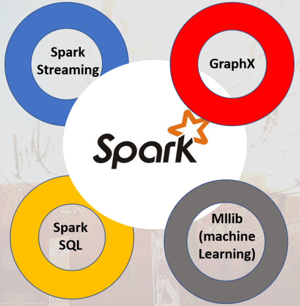
1. **Apache Spark Core:** Responsible for necessary functions such as Scheduling, task dispatching, input and output operations, fault recovery, etc.



Components of Spark

There are five main components of Apache Spark:

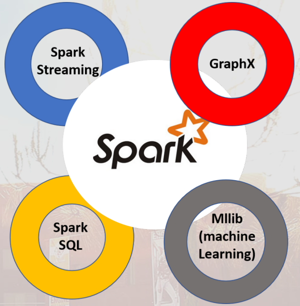
1. **Apache Spark Core:** Responsible for necessary functions such as Scheduling, task dispatching, input and output operations, fault recovery, etc.
2. **Spark Streaming:** Enables the processing of live data streams. Data can originate from many different sources, including Kafka, Kinesis, Flume, etc.



Components of Spark

There are five main components of Apache Spark:

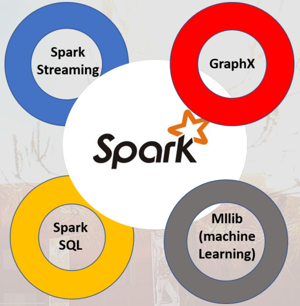
1. **Apache Spark Core:** Responsible for necessary functions such as Scheduling, task dispatching, input and output operations, fault recovery, etc.
2. **Spark Streaming:** Enables the processing of live data streams. Data can originate from many different sources, including Kafka, Kinesis, Flume, etc.
3. **Spark SQL:** Component to gather information about the structured data and how the data is processed.



Components of Spark

There are five main components of Apache Spark:

1. **Apache Spark Core:** Responsible for necessary functions such as Scheduling, task dispatching, input and output operations, fault recovery, etc.
2. **Spark Streaming:** Enables the processing of live data streams. Data can originate from many different sources, including Kafka, Kinesis, Flume, etc.
3. **Spark SQL:** Component to gather information about the structured data and how the data is processed.

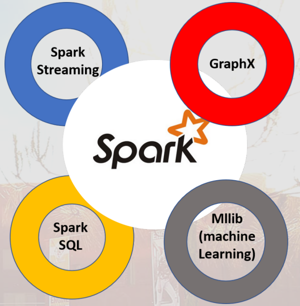


4. **Machine Learning Library (MLlib):** Consists of many machine learning algorithms.

Components of Spark

There are five main components of Apache Spark:

1. **Apache Spark Core:** Responsible for necessary functions such as Scheduling, task dispatching, input and output operations, fault recovery, etc.
2. **Spark Streaming:** Enables the processing of live data streams. Data can originate from many different sources, including Kafka, Kinesis, Flume, etc.
3. **Spark SQL:** Component to gather information about the structured data and how the data is processed.



4. **Machine Learning Library (MLlib):** Consists of many machine learning algorithms.
5. **GraphX:** Facilitating graph analytics tasks and graph-parallel computation.

Spark Architecture

Before diving deep into how Apache Spark works, lets understand the basics terminologies of Apache Spark:

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.

Spark Architecture

Before diving deep into how Apache Spark works, lets understand the basics terminologies of Apache Spark:

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- **Stages:** Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.

Spark Architecture

Before diving deep into how Apache Spark works, lets understand the basics terminologies of Apache Spark:

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- **Stages:** Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- **Tasks:** Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).

Spark Architecture

Before diving deep into how Apache Spark works, lets understand the basics terminologies of Apache Spark:

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- **Stages:** Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- **Tasks:** Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- **DAG:** DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.

Spark Architecture

Before diving deep into how Apache Spark works, lets understand the basics terminologies of Apache Spark:

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- **Stages:** Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- **Tasks:** Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- **DAG:** DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- **Executor:** The process responsible for executing a task.

Spark Architecture

Before diving deep into how Apache Spark works, lets understand the basics terminologies of Apache Spark:

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- **Stages:** Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- **Tasks:** Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- **DAG:** DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- **Executor:** The process responsible for executing a task.
- **Master:** The machine on which the Driver program runs

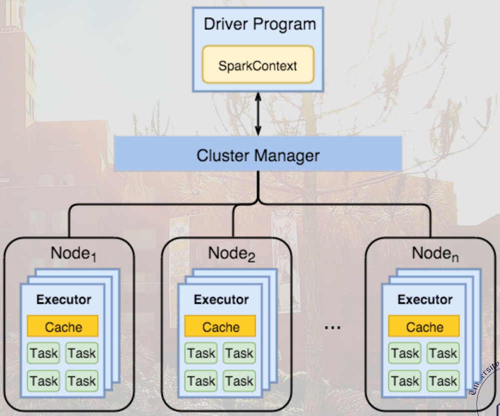
Spark Architecture

Before diving deep into how Apache Spark works, lets understand the basics terminologies of Apache Spark:

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- **Stages:** Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- **Tasks:** Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- **DAG:** DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- **Executor:** The process responsible for executing a task.
- **Master:** The machine on which the Driver program runs
- **Slave:** The machine on which the Executor program runs.

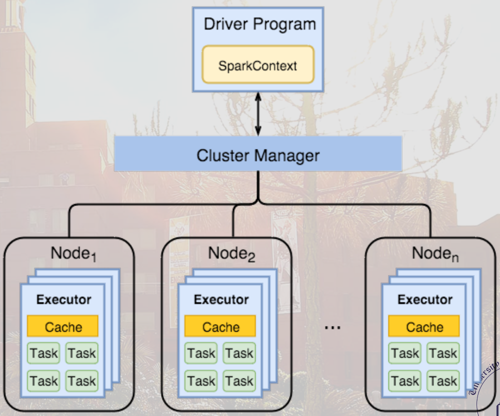
Spark Architecture

1. **Spark Driver:**
   - Separate process to execute user applications.
   - Creates SparkContext to schedule jobs execution and negotiate with cluster manager.



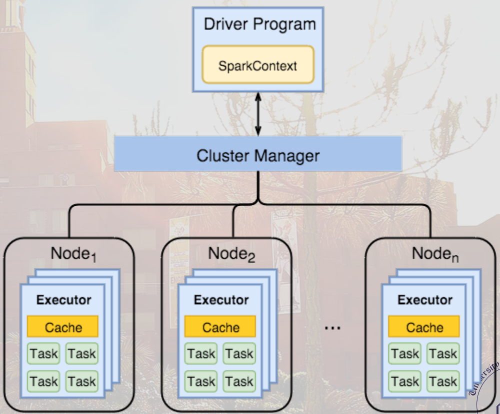
Spark Architecture

1. **Spark Driver:**
   - Separate process to execute user applications.
   - Creates SparkContext to schedule jobs execution and negotiate with cluster manager.
2. **Executors:**
   - Run tasks scheduled by driver



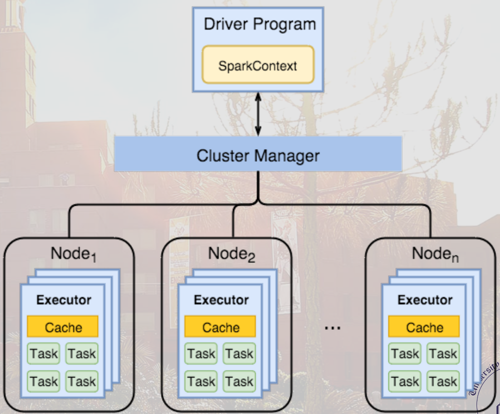
Spark Architecture

1. **Spark Driver:**
   - Separate process to execute user applications.
   - Creates SparkContext to schedule jobs execution and negotiate with cluster manager.
2. **Executors:**
   - Run tasks scheduled by driver
   - Store computation results in memory, on disk or off-heap



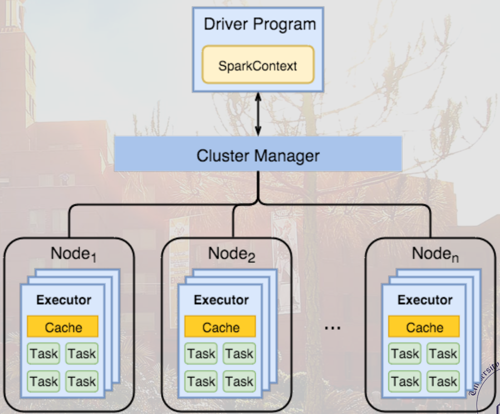
Spark Architecture

1. **Spark Driver:**
   - Separate process to execute user applications.
   - Creates SparkContext to schedule jobs execution and negotiate with cluster manager.
2. **Executors:**
   - Run tasks scheduled by driver
   - Store computation results in memory, on disk or off-heap
   - Interact with storage systems



Spark Architecture

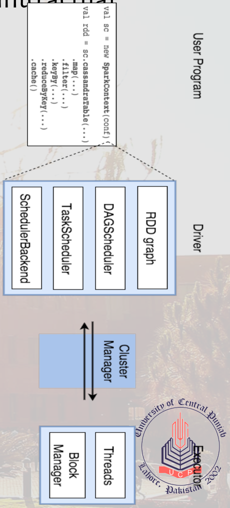
1. **Spark Driver:**
   - Separate process to execute user applications.
   - Creates SparkContext to schedule jobs execution and negotiate with cluster manager.
2. **Executors:**
   - Run tasks scheduled by driver
   - Store computation results in memory, on disk or off-heap
   - Interact with storage systems
3. **Cluster Manager:**
   - YARN or Spark Standalone or Mesos



Spark Architecture

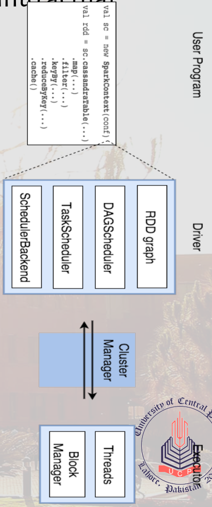
Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster:

1. **SparkContext:** It Represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.



Spark Architecture

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

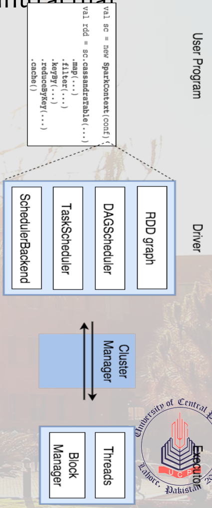


1. **SparkContext:** It Represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
2. **DAGScheduler:** It computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks.

Spark Architecture

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

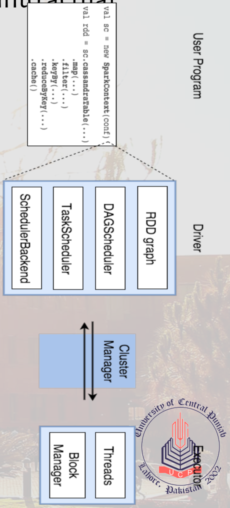
1. **SparkContext:** It Represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
2. **DAGScheduler:** It computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks.
3. **TaskScheduler:** Responsible for sending tasks to the cluster, running them, retrying if there are failures.



Spark Architecture

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

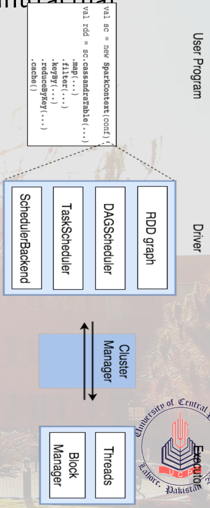
1. **SparkContext:** It Represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
2. **DAGScheduler:** It computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks.
3. **TaskScheduler:** Responsible for sending tasks to the cluster, running them, retrying if there are failures.
4. **SchedulerBackend:** It backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local).



Spark Architecture

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

1. **SparkContext:** It Represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
2. **DAGScheduler:** It computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks.
3. **TaskScheduler:** Responsible for sending tasks to the cluster, running them, retrying if there are failures.
4. **SchedulerBackend:** It backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local).
5. **BlockManager:** Provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap).



RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure.
- Provides API for:
  1. Manipulating the collection of elements (transformations and materialization).

RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure.
- Provides API for:
  1. Manipulating the collection of elements (transformations and materialization).
  2. Persisting intermediate results in memory for later reuse.

RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure.
- Provides API for:
  1. Manipulating the collection of elements (transformations and materialization).
  2. Persisting intermediate results in memory for later reuse.
  3. Controlling partitioning to optimize data placement.

RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure.
- Provides API for:
  1. Manipulating the collection of elements (transformations and materialization).
  2. Persisting intermediate results in memory for later reuse.
  3. Controlling partitioning to optimize data placement.
- Can be created through deterministic operation:
  1. from storage (distributed file system, database, plain file).

RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure.
- Provides API for:
  1. Manipulating the collection of elements (transformations and materialization).
  2. Persisting intermediate results in memory for later reuse.
  3. Controlling partitioning to optimize data placement.
- Can be created through deterministic operation:
  1. from storage (distributed file system, database, plain file).
  2. from another RDD.

RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure.
- Provides API for:
  1. Manipulating the collection of elements (transformations and materialization).
  2. Persisting intermediate results in memory for later reuse.
  3. Controlling partitioning to optimize data placement.
- Can be created through deterministic operation:
  1. from storage (distributed file system, database, plain file).
  2. from another RDD.
- Stores information about parent RDDs:
  1. for execution optimization and operations pipelining.

RDD: Resilient Distributed Dataset

- A fault-tolerant, immutable, parallel data structure.
- Provides API for:
  1. Manipulating the collection of elements (transformations and materialization).
  2. Persisting intermediate results in memory for later reuse.
  3. Controlling partitioning to optimize data placement.
- Can be created through deterministic operation:
  1. from storage (distributed file system, database, plain file).
  2. from another RDD.
- Stores information about parent RDDs:
  1. for execution optimization and operations pipelining.
  2. to recompute the data in case of failure.

RDD Operations

- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).

RDD Operations

- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).
  3. Apply aggregation function to the whole dataset (groupBy, sortBy).

RDD Operations

- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).
  3. Apply aggregation function to the whole dataset (groupBy, sortBy).
  4. Introduce dependencies between RDDs to form DAG.

RDD Operations

- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).
  3. Apply aggregation function to the whole dataset (groupBy, sortBy).
  4. Introduce dependencies between RDDs to form DAG.
  5. Provide functionality for repartitioning (repartition, partitionBy).

RDD Operations

- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).
  3. Apply aggregation function to the whole dataset (groupBy, sortBy).
  4. Introduce dependencies between RDDs to form DAG.
  5. Provide functionality for repartitioning (repartition, partitionBy).
- **Actions:**
  1. Trigger job execution

RDD Operations

- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).
  3. Apply aggregation function to the whole dataset (groupBy, sortBy).
  4. Introduce dependencies between RDDs to form DAG.
  5. Provide functionality for repartitioning (repartition, partitionBy).
- **Actions:**
  1. Trigger job execution
  2. Used to materialize computation results

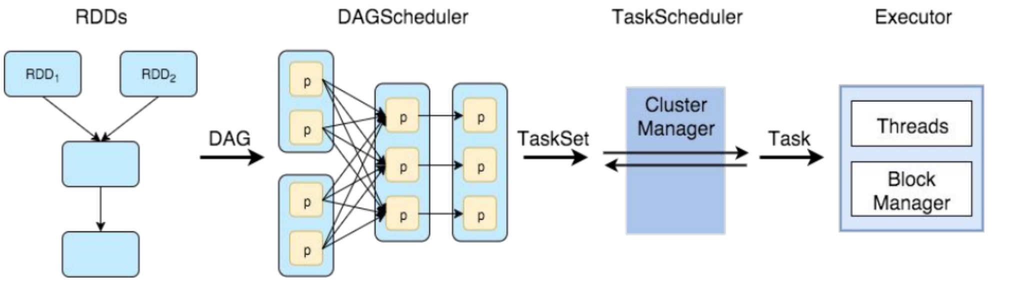
RDD Operations

- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).
  3. Apply aggregation function to the whole dataset (groupBy, sortBy).
  4. Introduce dependencies between RDDs to form DAG.
  5. Provide functionality for repartitioning (repartition, partitionBy).
- **Actions:**
  1. Trigger job execution
  2. Used to materialize computation results
- **Extra: persistence:**
  1. Explicitly store RDDs in memory, on disk or off-heap (cache, persist)

RDD Operations

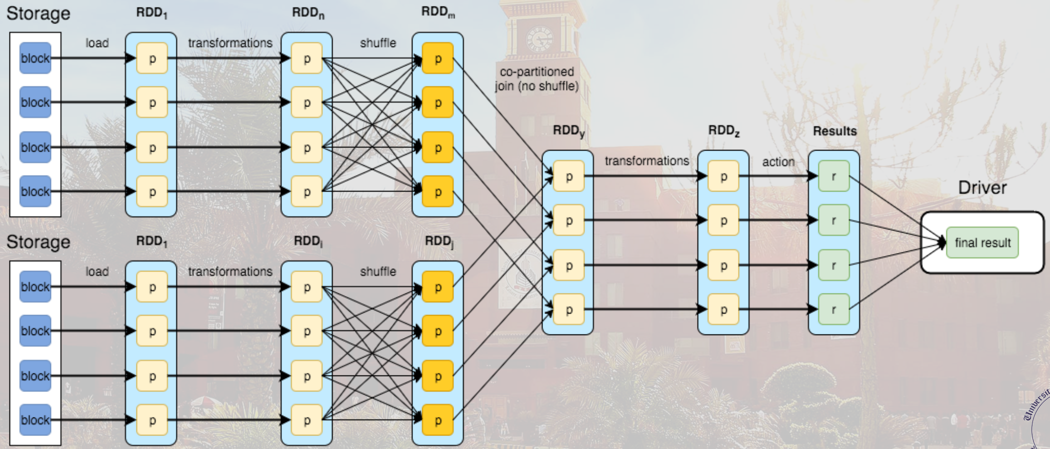
- **Transformations:**
  1. It is a function that produces new RDD from the existing RDDs. The newly created RDDs can not be reverted , so they are Acyclic.
  2. Apply user function (map(), filter(), union(), Cartesian(), etc..) to every element in a partition (or to the whole partition).
  3. Apply aggregation function to the whole dataset (groupBy, sortBy).
  4. Introduce dependencies between RDDs to form DAG.
  5. Provide functionality for repartitioning (repartition, partitionBy).
- **Actions:**
  1. Trigger job execution
  2. Used to materialize computation results
- **Extra: persistence:**
  1. Explicitly store RDDs in memory, on disk or off-heap (cache, persist)
  2. Check-pointing for truncating RDD lineage.

Execution Flow



RDDs	DAGScheduler	TaskScheduler	Executor
rdd1.**join**(rdd2) .**groupBy**(...) .**filter**(...)	splits graph into *stages* of tasks	launches tasks via cluster manager	executes tasks
	submits each stage as ready	retries failed or struggling tasks	stores and serves blocks

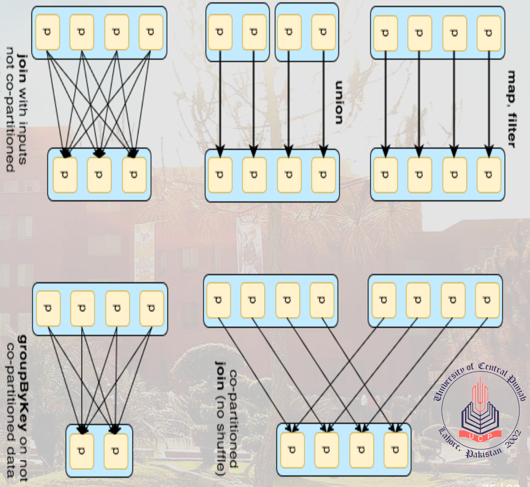
Direct Acyclic Graph (DAG)



Narrow and Wide Transformation

Transformations create dependencies between RDDs and here we can see different types of them.

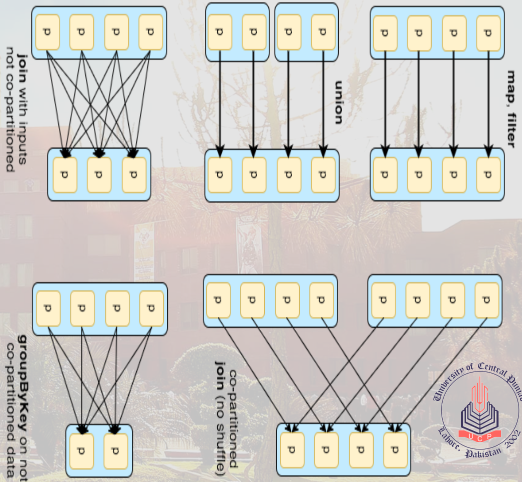
1. **Narrow (pipelineable):** each partition of the parent RDD is used by at most one partition of the child RDD.



Narrow and Wide Transformation

Transformations create dependencies between RDDs and here we can see different types of them.

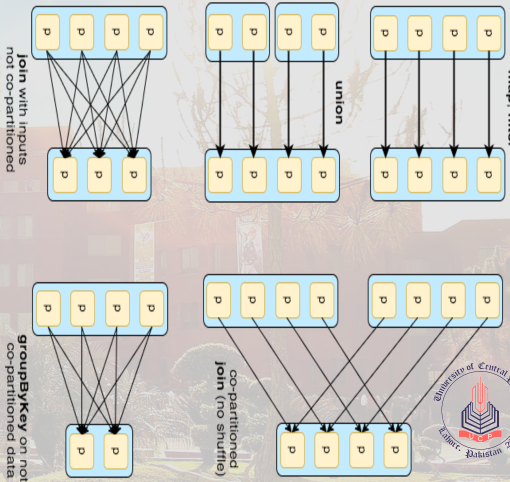
1. **Narrow (pipelineable):** each partition of the parent RDD is used by at most one partition of the child RDD.
2. Allow for pipelined execution on one cluster node



Narrow and Wide Transformation

Transformations create dependencies between RDDs and here we can see different types of them.

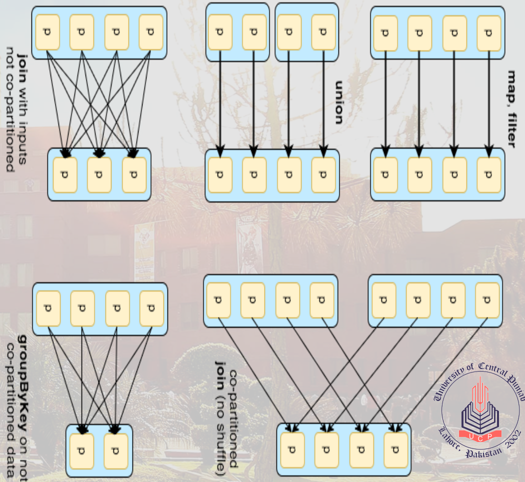
1. **Narrow (pipelineable):** each partition of the parent RDD is used by at most one partition of the child RDD.
2. Allow for pipelined execution on one cluster node
3. Failure recovery is more efficient as only lost parent partitions need to be recomputed.



Narrow and Wide Transformation

Transformations create dependencies between RDDs and here we can see different types of them.

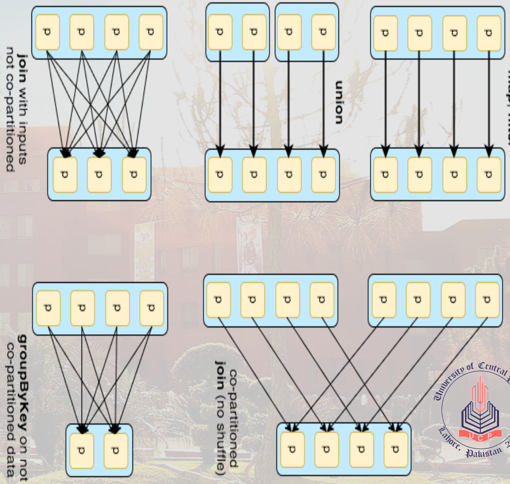
1. **Narrow (pipelineable):** each partition of the parent RDD is used by at most one partition of the child RDD.
2. Allow for pipelined execution on one cluster node
3. Failure recovery is more efficient as only lost parent partitions need to be recomputed.
4. **Wide (shuffle):** Multiple child partitions may depend on one parent partition.



Narrow and Wide Transformation

Transformations create dependencies between RDDs and here we can see different types of them.

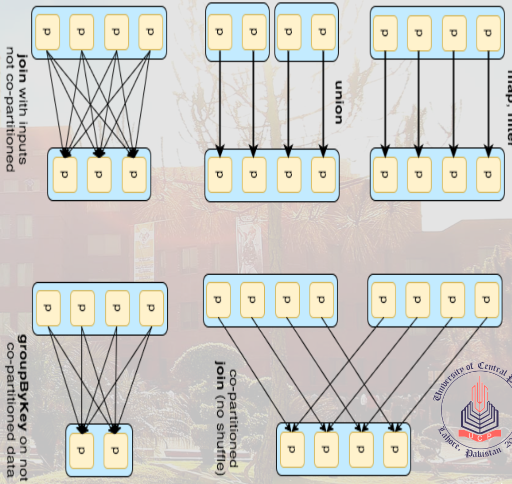
1. **Narrow (pipelineable):** each partition of the parent RDD is used by at most one partition of the child RDD.
2. Allow for pipelined execution on one cluster node
3. Failure recovery is more efficient as only lost parent partitions need to be recomputed.
4. **Wide (shuffle):** Multiple child partitions may depend on one parent partition.
5. Require data from all parent partitions to be available and to be shuffled across the nodes.



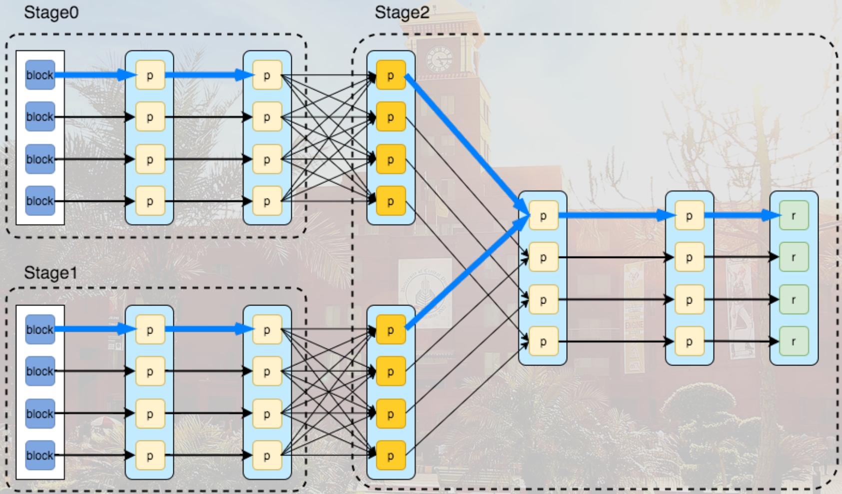
Narrow and Wide Transformation

Transformations create dependencies between RDDs and here we can see different types of them.

1. **Narrow (pipelineable):** each partition of the parent RDD is used by at most one partition of the child RDD.
2. Allow for pipelined execution on one cluster node
3. Failure recovery is more efficient as only lost parent partitions need to be recomputed.
4. **Wide (shuffle):** Multiple child partitions may depend on one parent partition.
5. Require data from all parent partitions to be available and to be shuffled across the nodes.
6. If some partition is lost from all the ancestors a complete recomputation is needed.



Narrow and Wide Transformation



Thank You