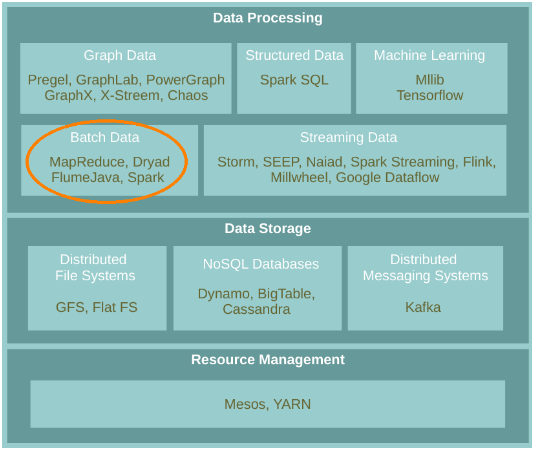
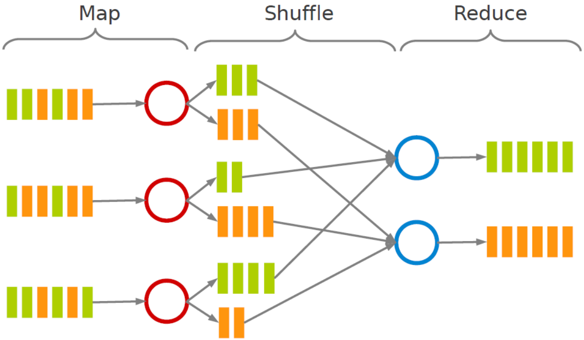
Parallel Processing - Spark

Amir H. Payberah
payberah@kth.se
2025-09-09

Where Are We?

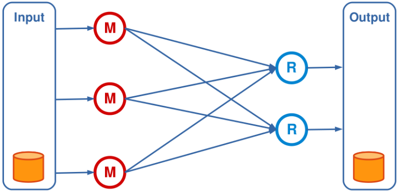

Data Processing
Graph Data
Pregel, GraphLab, PowerGraph
GraphX, X-Streem, Chaos
Structured Data
Spark SQL
Machine Learning
Mllib
Tensorflow
Batch Data
MapReduce, Dryad
FlumeJava, Spark
Streaming Data
Storm, SEEP, Naiad, Spark Streaming, Flink,
Millwheel, Google Dataflow
Data Storage
Distributed
File Systems
GFS, Flat FS
NoSQL Databases
Dynamo, BigTable,
Cassandra
Distributed
Messaging Systems
Kafka
Resource Management
Mesos, YARN

MapReduce Reminder



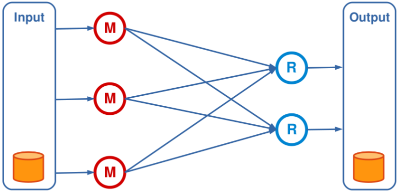
Motivation (1/2)

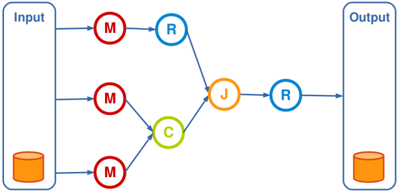
- Acyclic data flow from stable storage to stable storage.



Motivation (1/2)

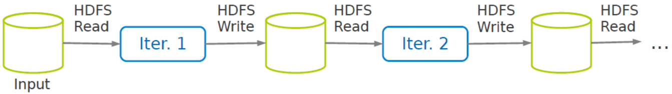
- Acyclic data flow from stable storage to stable storage.



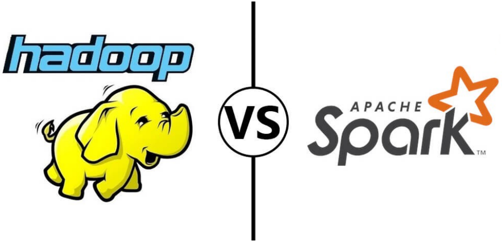


Motivation (2/2)

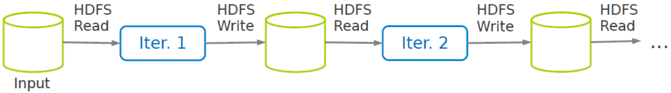
- MapReduce is expensive (slow), i.e., always goes to disk and HDFS.



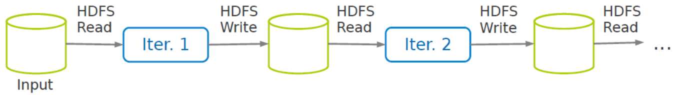
So, Let's Use Spark

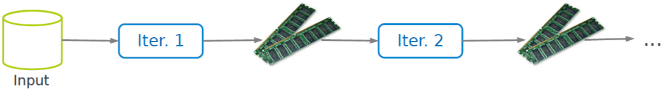


Spark vs. MapReduce (1/2)

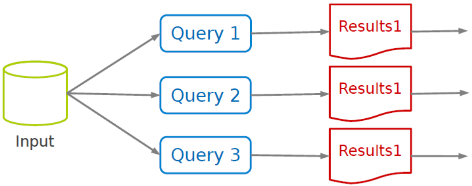


Spark vs. MapReduce (1/2)

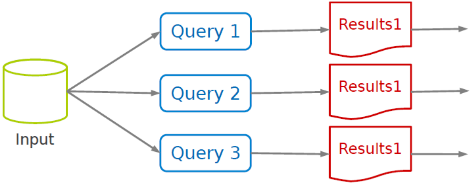


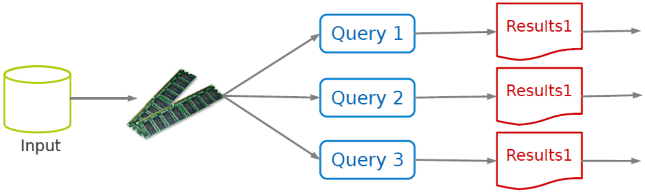


Spark vs. MapReduce (2/2)



Spark vs. MapReduce (2/2)

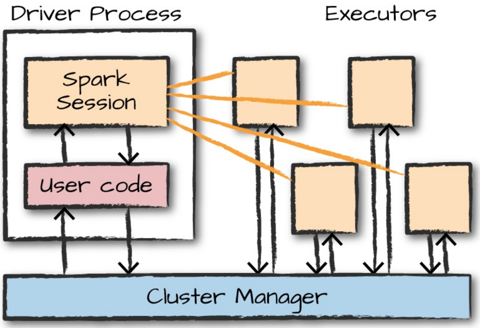




Spark Application

Spark Applications Architecture

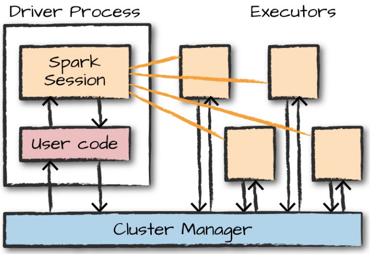
- Spark applications consist of
  - A driver process
  - A set of executor processes



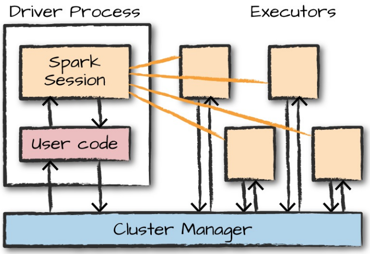
[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Driver Process

- The heart of a Spark application
- Runs the `main()` function



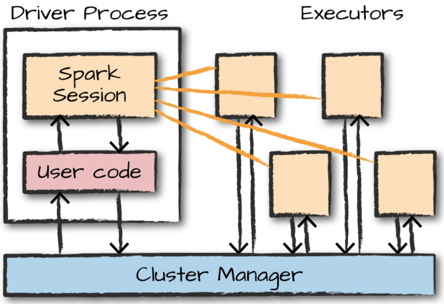
Driver Process



- The heart of a Spark application
- Runs the `main()` function
- Responsible for three things:
  - Maintaining information about the Spark application
  - Responding to a user's program or input
  - Analyzing, distributing, and scheduling work across the executors

Executors

- Executing code assigned to it by the driver
- Reporting the state of the computation on that executor back to the driver



SparkSession

- A driver process that controls a Spark application.

```
SparkSession.builder.master(master).appName(appName).getOrCreate()
```

SparkSession

- A driver process that controls a Spark application.
- A one-to-one correspondence between a SparkSession and a Spark application.

```
SparkSession.builder.master(master).appName(appName).getOrCreate()
```

SparkSession

- A driver process that controls a Spark application.
- A one-to-one correspondence between a `SparkSession` and a Spark application.
- Available in console shell as `spark`.

```
SparkSession.builder.master(master).appName(appName).getOrCreate()
```

SparkContext

- The entry point for low-level API functionality.
- You access it through the SparkSession.

```
val conf = new SparkConf().setMaster(master).setAppName(appName)
new SparkContext(conf)
```

SparkContext

- The entry point for low-level API functionality.
- You access it through the SparkSession.
- Available in console shell as sc.

```
val conf = new SparkConf().setMaster(master).setAppName(appName)
new SparkContext(conf)
```

SparkSession vs. SparkContext

- Prior to Spark 2.0.0, a the spark driver program uses SparkContext to connect to the cluster.

- In order to use APIs of SQL, Hive and streaming, separate SparkContexts should to be created.

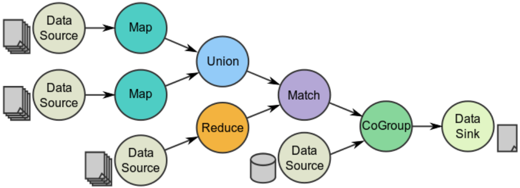
SparkSession vs. SparkContext

- Prior to Spark 2.0.0, a the spark driver program uses SparkContext to connect to the cluster.
- In order to use APIs of SQL, Hive and streaming, separate SparkContexts should to be created.
- SparkSession provides access to all the spark functionalities that SparkContext does, e.g., SQL, Hive and streaming.
- SparkSession internally has a SparkContext for actual computation.

Programming Model

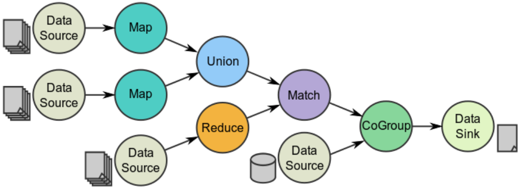
Spark Programming Model

- Job is described based on directed acyclic graphs (DAG) data flow.



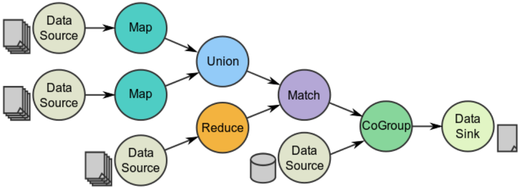
Spark Programming Model

- Job is described based on directed acyclic graphs (DAG) data flow.
- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.



Spark Programming Model

- Job is described based on directed acyclic graphs (DAG) data flow.
- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Parallelizable operators



Resilient Distributed Datasets (RDD) (1/3)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.
  - Like a `LinkedList <MyObjects>`

Resilient Distributed Datasets (RDD) (2/3)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.

Resilient Distributed Datasets (RDD) (3/3)

- RDDs were the primary API in the Spark 1.x series.
- They are not commonly used in the Spark 2.x series.
- Virtually all Spark code you run, compiles down to an RDD.

Types of RDDs

- Two types of RDDs:
  - Generic RDD
  - Key-value RDD
- Both represent a collection of objects.
- Key-value RDDs have special operations, such as aggregation, and a concept of custom partitioning by key.

Creating RDDs

Creating RDDs - Parallelized Collections

- Use the `parallelize` method on a `SparkContext`.
- This turns a single node collection into a parallel collection.
- You can also explicitly state the number of partitions.
- In the console shell, you can either use `sc` or `spark.sparkContext`

Creating RDDs - Parallelized Collections

- Use the `parallelize` method on a `SparkContext`.
- This turns a single node collection into a parallel collection.
- You can also explicitly state the number of partitions.
- In the console shell, you can either use `sc` or `spark.sparkContext`

```
val numsCollection = Array(1, 2, 3)
val nums = sc.parallelize(numsCollection)
```

Creating RDDs - Parallelized Collections

- Use the parallelize method on a SparkContext.
- This turns a single node collection into a parallel collection.
- You can also explicitly state the number of partitions.
- In the console shell, you can either use sc or spark.sparkContext

```
val numsCollection = Array(1, 2, 3)
val nums = sc.parallelize(numsCollection)
```

```
val wordsCollection = "take it easy, this is a test".split(" ")
val words = spark.sparkContext.parallelize(wordsCollection, 2)
```

Creating RDDs - External Datasets

- Create RDD from an external storage.
  - E.g., local file system, HDFS, Cassandra, HBase, Amazon S3, etc.
- Text file RDDs can be created using `textFile` method.

```
val myFile1 = sc.textFile("file.txt")
val myFile2 = sc.textFile("hdfs://namenode:9000/path/file")
```

RDD Operations

RDD Operations

- RDDs support two types of operations:
  - Transformations: allow us to build the logical plan
  - Actions: allow us to trigger the computation

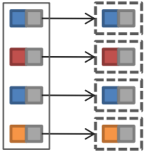
Transformations

Transformations

- Create a new RDD from an existing one.
- All transformations are lazy.
    - Not compute their results right away.
    - Remember the transformations applied to the base dataset.
    - They are only computed when an action requires a result to be returned to the driver program.

Generic RDD Transformations

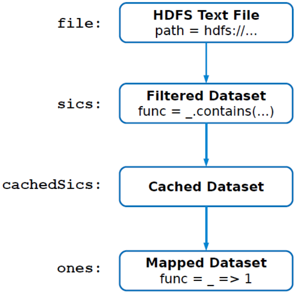
- `map` applies a given function on each RDD record **independently**.



```
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x)
// 1, 4, 9
```

Lineage

- Lineage: transformations used to build an RDD.
- RDDs are stored as a chain of objects capturing the lineage of each RDD.




```scala
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

Convert each elements `x` of an RDD to `(x, 1)`?

`https://tinyurl.com/4bys7j7t`

Key-Value RDD Transformations

- In a (k, v) pairs, k is is the key, and v is the value.

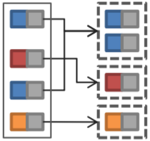
Key-Value RDD Transformations

- In a (k, v) pairs, k is is the key, and v is the value.
- To make a key-value RDD:
  - `map` over your current RDD to a basic key-value structure.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))
val keyword1 = words.map(word => (word, 1))
// (take,1), (it,1), (easy,,1), (this,1), (is,1), (a,1), (test,1)
```

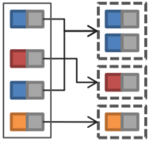
Key-Value RDD Transformations - Aggregation

- **Aggregate the values** associated with each key.



```
def addFunc(a:Int, b:Int) = a + b

val kvChars = ...
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,,1), ...

val grpChar = kvChars.groupByKey().map(row => (row._1, row._2.reduce(addFunc)))
// (t,5), (h,1), (,,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))
```

Key-Value RDD Transformations - Aggregation



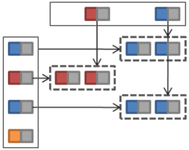
- Aggregate the values associated with each key.

```
def addFunc(a:Int, b:Int) = a + b

val kvChars = ...
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,,1), ...

val grpChar = kvChars.groupByKey().map(row => (row._1, row._2.reduce(addFunc)))
// (t,5), (h,1), (,,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))
```

```
val redChar = kvChars.reduceByKey(addFunc)
// (t,5), (h,1), (,,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))
```

Key-Value RDD Transformations - Join

- join performs an inner-join on the key.
- fullOtherJoin, leftOuterJoin, rightOuterJoin, and cartesian.



```
val keyedChars = ...
// (t,4), (h,6), (,,9), (e,8), (a,3), (i,5), (y,2), (s,7), (k,0)

val kvChars = ...
// (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,,1), ...

val joinedChars = kvChars.join(keyedChars)
// (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (t,(1,4)), (h,(1,6)), (,,(1,9)), (e,(1,8)), ...
```

Actions

Actions

- Transformations allow us to build up our logical transformation plan (lineage graph).
- We run an action to trigger the computation.
  - Instructs Spark to compute a result from a series of transformations.

RDD Actions (1/3)

- `collect` returns all the elements of the RDD as an array at the driver.

```
val nums = sc.parallelize(Array(1, 2, 3))

nums.collect()
// Array(1, 2, 3)
```

RDD Actions (2/3)

- `reduce` aggregates the elements of the dataset using a given function.
- The given function should be commutative and associative so that it can be computed correctly in parallel.

```
sc.parallelize(1 to 20).reduce(_ + _)
// 210
```

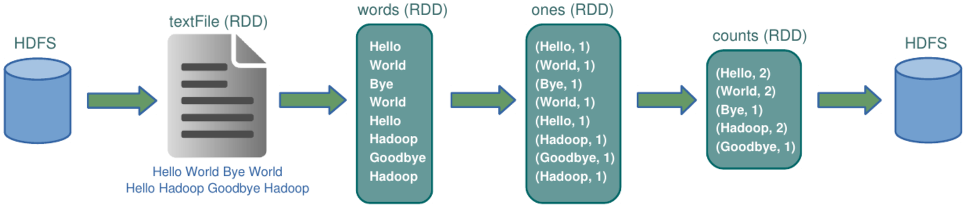
RDD Actions (3/3)

- `saveAsTextFile` writes the elements of an RDD as a text file.
  - Local filesystem, HDFS or any other Hadoop-supported file system.
- `saveAsObjectFile` explicitly writes key-value pairs.

```
val words = sc.parallelize("take it easy, this is a test".split(" "))

words.saveAsTextFile("file:/tmp/words")
```

Spark Word-Count

```
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



What Is The Problem?

```
val uni = sc.parallelize(Seq(("RISE", 1), ("KTH", 2)))
uni.foreach(println)
```

Cache and Checkpoints

Caching

- When you cache an RDD, each node stores any partitions of it that it computes in memory.
- An RDD that is not cached is re-evaluated each time an action is invoked on that RDD.
- A node reuses the cached RDD in other actions on that dataset.

Caching

- When you cache an RDD, each node stores any partitions of it that it computes in memory.
- An RDD that is not cached is re-evaluated each time an action is invoked on that RDD.
- A node reuses the cached RDD in other actions on that dataset.
- There are two functions for caching an RDD:
  - `cache` caches the RDD into memory
  - `persist(level)` can cache in memory, on disk, or off-heap memory

```
val words = sc.parallelize("take it easy, this is a test".split(" "))

words.cache()
```

Checkpointing

- `checkpoint` saves an RDD to disk.
- Checkpointed data is not removed after `SparkContext` is destroyed.
- When we reference a checkpointed RDD, it will derive from the checkpoint instead of the source data.

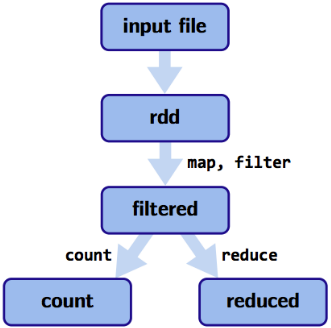
```
val words = sc.parallelize("take it easy, this is a test".split(" "))

sc.setCheckpointDir("/path/checkpointing")
words.checkpoint()
```

Execution Engine

More About Lineage

- A DAG representing the computations done on the RDD is called lineage graph.

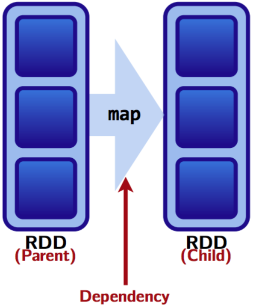
```
val rdd = sc.textFile(...)
val filtered = rdd.map(...).filter(...).persist()
val count = filtered.count()
val reduced = filtered.reduce()
```

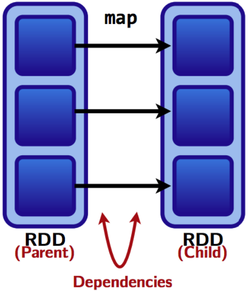


[https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies]

Dependencies

- RDD dependencies encode when data must move across network.

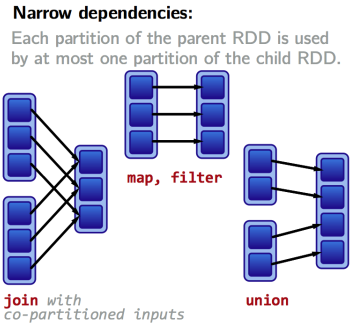




[https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies]

Two Types of Dependencies (1/2)

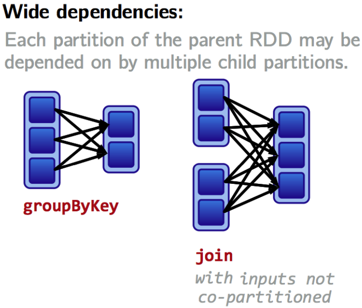
- Narrow transformations (dependencies)
  - Each input partition will contribute to only one output partition.
  - With narrow transformations, Spark can perform a pipelining



[https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies]

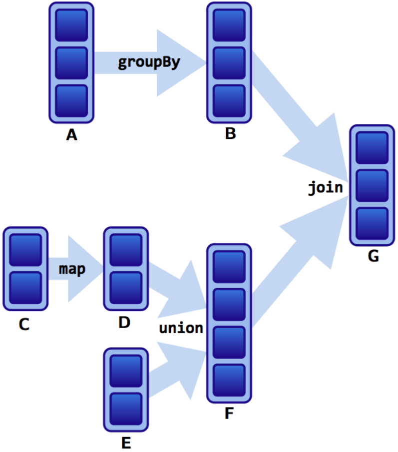
Two Types of Dependencies (2/2)

- Wide transformations (dependencies)
  - Each input partition will contribute to many output partition.
  - Usually referred to as a shuffle

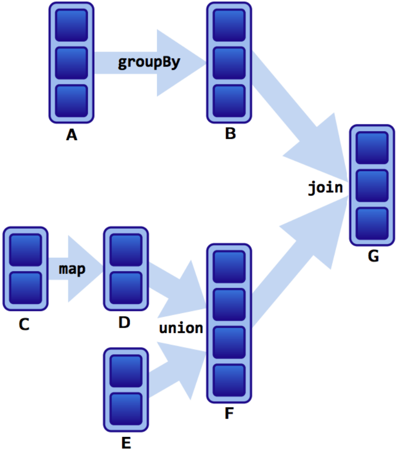


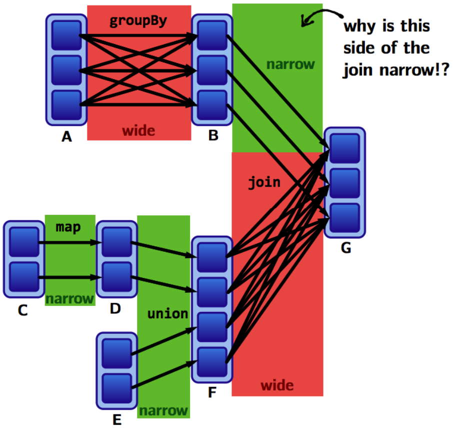
[https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies]

Example



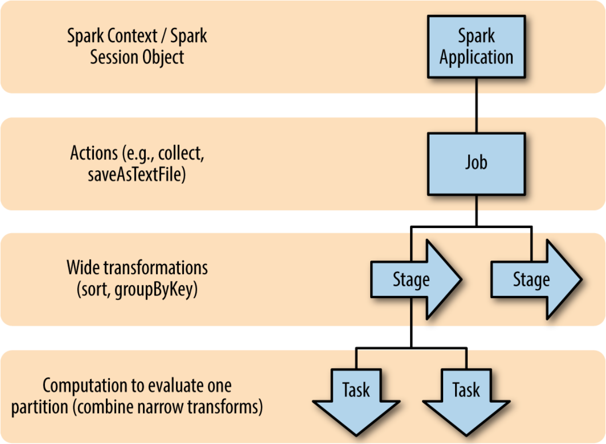
Example





[https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies]

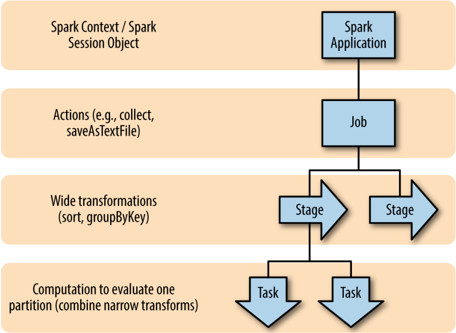
The Anatomy of a Spark Job



[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]

Jobs

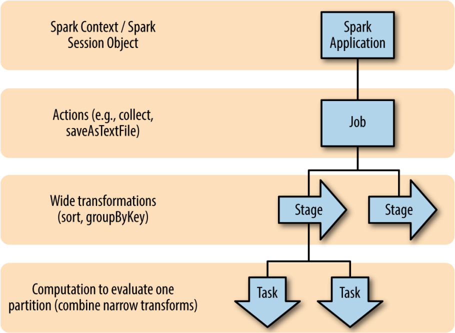
- A Spark job is the highest element of Spark's execution hierarchy.
  - Each Spark job corresponds to one action.
  - Each action is called by the driver program of a Spark application.



[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]

Stages

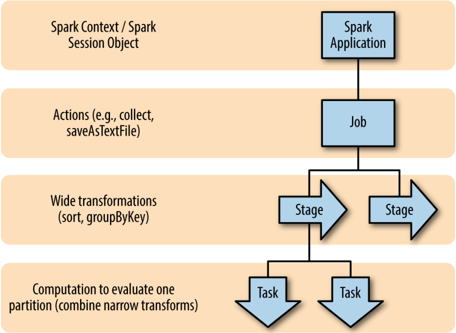
- Each job breaks down into a series of stages.
  - Stages in Spark represent groups of tasks that can be executed together.
  - Wide transformations define the breakdown of jobs into stages.



[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]

Tasks

- A stage consists of tasks, which are the smallest execution unit.
  - Each task represents one local computation.
  - All of the tasks in one stage execute the same code on a different piece of the data.



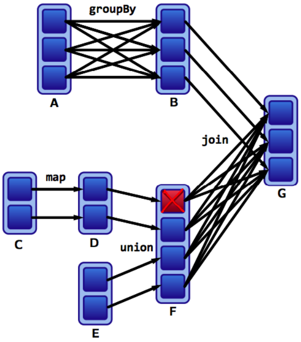
[H. Karau et al., High Performance Spark, O'Reilly Media, 2017]

Lineages and Fault Tolerance (1/2)

- No replication.
- Lineages are the key to fault tolerance in Spark.
- Recompute only the lost partitions of an RDD.

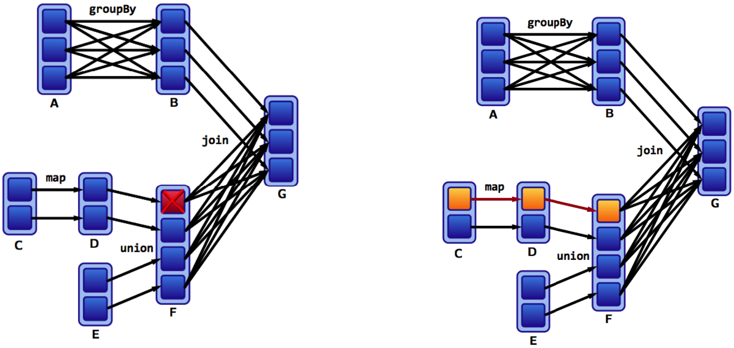
Lineages and Fault Tolerance (2/2)

- Assume one of the partitions fails.



Lineages and Fault Tolerance (2/2)

- Assume one of the partitions fails.
- We only have to recompute the data shown below to get back on track.



[https://github.com/rohgar/scala-spark-4/wiki/Wide-vs-Narrow-Dependencies]

Questions

- If Spark became the standard tool for public decision-making (e.g., healthcare, housing, employment), what risks of reinforcing inequities might arise?

Possible Answers

- Biased inputs $\rightarrow$ biased outputs

Possible Answers

- Biased inputs $\rightarrow$ biased outputs
  - If the input data reflects past discrimination (e.g., in jobs or housing), Spark will repeat those patterns at scale.

Possible Answers

- Biased inputs $\rightarrow$ biased outputs
  - If the input data reflects past discrimination (e.g., in jobs or housing), Spark will repeat those patterns at scale.
- Unfair results may look objective

Possible Answers

- Biased inputs → biased outputs
  - If the input data reflects past discrimination (e.g., in jobs or housing), Spark will repeat those patterns at scale.
- Unfair results may look objective
  - Because decisions come from data and code, they can appear neutral even when they reinforce existing inequities.

Possible Answers

- Biased inputs $\rightarrow$ biased outputs
  - If the input data reflects past discrimination (e.g., in jobs or housing), Spark will repeat those patterns at scale.
- Unfair results may look objective
  - Because decisions come from data and code, they can appear neutral even when they reinforce existing inequities.
- Big players control the system

Possible Answers

- Biased inputs → biased outputs
  - If the input data reflects past discrimination (e.g., in jobs or housing), Spark will repeat those patterns at scale.
- Unfair results may look objective
  - Because decisions come from data and code, they can appear neutral even when they reinforce existing inequities.
- Big players control the system
  - Large companies or agencies with the most resources shape how Spark is used, leaving smaller communities behind.

Questions

- Feminist Spark alternatives?

Possible Answers

- Transparency and provenance

Possible Answers

- Transparency and provenance
  - OpenLineage/Marquez: open standards for tracking pipeline provenance.

Possible Answers

- Transparency and provenance
  - OpenLineage/Marquez: open standards for tracking pipeline provenance.
- Lightweight and community-friendly

Possible Answers

- Transparency and provenance
  - OpenLineage/Marquez: open standards for tracking pipeline provenance.
- Lightweight and community-friendly
  - Dask: Python-native distributed system, runs on laptops or clusters, more accessible than Spark.

Summary

Summary

- RDD: a distributed memory abstraction
- Two types of operations: transformations and actions
- Lineage graph
- Caching
- Wide vs. narrow dependencies
- Alternatives: Openlineage, Marquez, Dask

References

- M. Zaharia et al., "Spark: The Definitive Guide", O'Reilly Media, 2018 - Chapters 2, 12, 13, and 14
- M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", USENIX NSDI, 2012.
- Dask: https://www.dask.org
- OpenLineage: https://openlineage.io
- Marquez: https://marquezproject.ai/

Questions?