Distributed platforms for Big Data

Prof. Carlo Ferrari
Michele Stecca, Ph.D.
a.y. 2023-2024
Data, data, data

2021 This Is What Happens In An Internet Minute

- Facebook: 1.4 Million Posts
- LinkedIn: 9,132 Connections Made
- YouTube: 21.1 Million Texts Sent
- Netflix: 28,000 Subscribers Watching
- Pinterest: 1.6 Million Spent Online
- Snapchat: 3.4 Million Snaps Created
- Instagram: 69 Million Stories Shared
- Tinder: 200,000 People Tweeting
- TikTok: 2 Million Views
- Twitch: 5,000 Downloads
- Amazon Echo: 932 Smart Audio Devices Shipped
- WhatsApp: 3 Million Images Viewed
- Gmail: 2 Million Emails Sent
- Twitter: 2 Million Swipes
- Google Play: 414,764 Apps Downloaded
- YouTube: 500 Hours Content Uploaded

Created By: @LoriLewis @OfficiallyChadd
Vertical vs. Horizontal scalability

**Scalability**

**Vertical scaling**

**Horizontal scaling**

Increase in number of machines

Increase in processing power
Design principles

(Data) Partitioning
Design principles

(Data) Replication
An architecture for Big Data

Distributed Filesystem
HDFS

HaDoop FileSystem Architecture

- Files are splitted into >64MB blocks
- Throughput optimized
- Write once – read many
- Failure resistant: manages failures using block replication
HDFS

Master & Slave architecture

Master Node

NameNode

Meta Data

Slave Node

DataNode

Application Data

Slave Node

DataNode

Application Data

Slave Node

DataNode

Application Data
HaDoop FileSystem Architecture: Name Node

- Cluster’s SPOF!
- Manages the file system
- Lists the files and blocks in which they are divided
- Manages strategies replication and block allocation
- Checks nodes’s reliability
HDFS

HaDoop FileSystem Architecture: DataNode

- Manages the storage and the client’s requests
- Sends Heartbeat to NameNode
An architecture for Big Data

Move computation close to data!!!

Distributed Computational Model
+ Execution Engine

Distributed Filesystem (HDFS)
Hadoop versions

HADOOP 1.0

MapReduce
(cluster resource management & data processing)

HDFS
(redundant, reliable storage)

HADOOP 2.0

MapReduce
(data processing)

YARN
(cluster resource management)

HDFS
(redundant, reliable storage)

Others
(data processing)
Cluster management: Yarn

Yet Another Resource Negotiator architecture:

Resource Manager:
- One per cluster – global view
- No static resource partitioning
- Handle Job request
- Find a container to Application Manager
An architecture for Big Data

- Distributed Computational Model + Execution Engine: Map Reduce
- Resource Manager (YARN)
- Distributed Filesystem (HDFS)
An architecture for Big Data

<table>
<thead>
<tr>
<th>Resource Manager (YARN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Computational Model + Execution Engine (Map Reduce)</td>
</tr>
<tr>
<td>Applications («Pure» MR Apps, SQL, Machine Learning, Graphs, Streaming, etc.)</td>
</tr>
<tr>
<td>Distributed Filesystem (HDFS)</td>
</tr>
</tbody>
</table>
Map Reduce limitations

- Not so flexible from a programmer point of view
- Not so efficient
- ...

Iterative jobs involve a lot of disk I/O for each repetition

Disk I/O is very slow!
From Hadoop/MapReduce to Spark

Applications («Pure» MR Apps, SQL, Machine Learning, Graphs, Streaming, etc.)

Distributed Computational Model
+ Execution Engine: Map Reduce

Resource Manager (YARN)

Distributed Filesystem (HDFS)
From Hadoop/MapReduce to Spark

Applications («Pure» Spark Apps, SQL, Machine Learning, Graphs, Streaming, etc.)

Distributed Computational Model + Execution Engine: Spark

Resource Manager (YARN)

Distributed Filesystem (HDFS)
From Hadoop/MapReduce to Spark

- Distributed Filesystem (HDFS)
- Applications (Simple APIs, SQL, Machine Learning, Graphs, Streaming, etc.)
- Distributed Computational Model + Execution Engine: Spark
- Resource Manager (YARN), Mesos, Kubernetes
From Hadoop/MapReduce to Spark

Applications (Simple APIs, SQL, Machine Learning, Graphs, Streaming, etc.)

Distributed Computational Model
+ Execution Engine: Spark

Resource Manager (YARN)  |  Mesos  |  Kubernetes

HDFS  |  S3  |  Cassandra  |  ...

---

Diagram showing server racks and hard drives, likely representing a cloud or big data infrastructure setup.
MapReduce vs. Spark

- Spark is in-memory
- Less expensive shuffles
- There are many more primitives
- It supports Java, Scala, Python & R
- Interactive shells are available
- Generalized patterns
  - unified engine for many use cases
- Lazy evaluation of the lineage graph
  - reduces wait states, better pipelining
- Lower overhead for starting jobs
Spark ecosystem

- Spark SQL
- Spark Streaming
- MLlib (machine learning)
- GraphX (graph)

Apache Spark
Spark (Deployment)
• Resilient Distributed Datasets (RDDs) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

• 2 types of operations on RDDs:
  • transformations and actions
    • transformations are lazy (not computed immediately)
  • however, an RDD can be persisted into storage in memory or disk
Some Primitives

<table>
<thead>
<tr>
<th>transformation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(func)</code></td>
<td>return a new distributed dataset formed by passing each element of the source through a function <code>func</code></td>
</tr>
<tr>
<td><code>filter(func)</code></td>
<td>return a new dataset formed by selecting those elements of the source on which <code>func</code> returns true</td>
</tr>
<tr>
<td><code>flatMap(func)</code></td>
<td>similar to map, but each input item can be mapped to 0 or more output items (so <code>func</code> should return a <code>Seq</code> rather than a single item)</td>
</tr>
<tr>
<td><code>sample(withReplacement, fraction, seed)</code></td>
<td>sample a fraction <code>fraction</code> of the data, with or without replacement, using a given random number generator seed</td>
</tr>
<tr>
<td><code>union(otherDataset)</code></td>
<td>return a new dataset that contains the union of the elements in the source dataset and the argument</td>
</tr>
<tr>
<td><code>distinct([numTasks]))</code></td>
<td>return a new dataset that contains the distinct elements of the source dataset</td>
</tr>
<tr>
<td>transformation</td>
<td>description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>groupByKey([numTasks])</code></td>
<td>when called on a dataset of ((k, v)) pairs, returns a dataset of ((k, \text{seq}[v])) pairs</td>
</tr>
<tr>
<td><code>reduceByKey(func, [numTasks])</code></td>
<td>when called on a dataset of ((k, v)) pairs, returns a dataset of ((k, v)) pairs where the values for each key are aggregated using the given reduce function</td>
</tr>
<tr>
<td><code>sortByKey([ascending], [numTasks])</code></td>
<td>when called on a dataset of ((k, v)) pairs where (k) implements ordered, returns a dataset of ((k, v)) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument</td>
</tr>
<tr>
<td><code>join(otherDataset, [numTasks])</code></td>
<td>when called on datasets of type ((k, v)) and ((k, w)), returns a dataset of ((k, (v, w))) pairs with all pairs of elements for each key</td>
</tr>
<tr>
<td><code>cogroup(otherDataset, [numTasks])</code></td>
<td>when called on datasets of type ((k, v)) and ((k, w)), returns a dataset of ((k, \text{seq}[v], \text{seq}[w])) tuples — also called groupWith</td>
</tr>
<tr>
<td><code>cartesian(otherDataset)</code></td>
<td>when called on datasets of types (T) and (U), returns a dataset of ((T, U)) pairs (all pairs of elements)</td>
</tr>
</tbody>
</table>
### Some Primitives

<table>
<thead>
<tr>
<th>action</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>reduce</strong>(func)</td>
<td>aggregate the elements of the dataset using a function <code>func</code> (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel</td>
</tr>
<tr>
<td><strong>collect</strong>()</td>
<td>return all the elements of the dataset as an array at the driver program — usually useful after a filter or other operation that returns a sufficiently small subset of the data</td>
</tr>
<tr>
<td><strong>count</strong>()</td>
<td>return the number of elements in the dataset</td>
</tr>
<tr>
<td><strong>first</strong>()</td>
<td>return the first element of the dataset — similar to <code>take(1)</code></td>
</tr>
<tr>
<td><strong>take</strong>(n)</td>
<td>return an array with the first <code>n</code> elements of the dataset — currently not executed in parallel, instead the driver program computes all the elements</td>
</tr>
<tr>
<td><strong>takeSample</strong>(withReplacement, fraction, seed)</td>
<td>return an array with a random sample of <code>num</code> elements of the dataset, with or without replacement, using the given random number generator seed</td>
</tr>
</tbody>
</table>
## Some Primitives

<table>
<thead>
<tr>
<th>action</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>saveAsTextFile(path)</code></td>
<td>write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element to convert it to a line of text in the file.</td>
</tr>
<tr>
<td><code>saveAsSequenceFile(path)</code></td>
<td>write the elements of the dataset as a Hadoop sequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's <code>Writable</code> interface or are implicitly convertible to <code>Writable</code> (Spark includes conversions for basic types like <code>Int</code>, <code>Double</code>, <code>String</code>, etc.).</td>
</tr>
<tr>
<td><code>countByKey()</code></td>
<td>only available on RDDs of type <code>(k, v)</code>. Returns a 'Map' of <code>(k, Int)</code> pairs with the count of each key.</td>
</tr>
<tr>
<td><code>foreach(func)</code></td>
<td>run a function <code>func</code> on each element of the dataset—it usually done for side effects such as updating an accumulator variable or interacting with external storage systems</td>
</tr>
</tbody>
</table>
Some Primitives

- Spark can *persist* (or cache) a dataset in memory across operations.
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster.
- The cache is *fault-tolerant*: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.
Spark SQL

• Spark SQL is a Spark module for structured data processing

• Uses more information about the structure of both the data and the computation being performed

• Spark SQL uses this extra information to perform extra optimizations

• Integrated with Hive metastore
Spark SQL: the Dataframe abstraction

- The **DataFrame API** provides a **higher-level abstraction**, allowing you to use a query language to manipulate data. In fact, you can use **SQL**, as well.
- This code does essentially the same thing the previous RDD code does. Look how much easier it is to read.
- You have probably met DataFrames already in Python or R
Spark SQL: the Dataframe abstraction

• It provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL’s optimized execution engine.

• **It is conceptually equivalent to a table in a relational database or a data frame** in R/Python, but with richer optimizations under the hood.

• DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
Spark SQL: the Dataframe abstraction

Data Sources supported by DataFrames

built-in
- Parquet
- JDBC
- { JSON }
- Hive
- MySQL
- HDFS
- Amazon S3
- PostgreSQL

external
- AVRO
- CSV
- dBase
- Apache HBase
- elastic search
- Cassandra
- Amazon Redshift
- and more ...
Example Optimization

users.join(events, users("id") === events("uid"))
  .filter(events("date") > "2015-01-01")

logical plan

  filter

  join

  scan (users)  scan (events)

optimized plan

  join

  scan (users)  filter

  scan (events)

optimized plan

  join

  scan (users)  filter scan (events)

Catalyst pushes the filter into the data source
e.g.: SELECT * FROM events WHERE user_id = ___
Spark SQL: JOIN Operations

(Distributed) JOIN Types
- Shuffle JOIN
- Broadcast JOIN
- Merge Sort JOIN
- Skew JOIN

JOIN Optimization parameters

`spark.sql.adaptive.enabled` - if this option is set to True Spark will make use of the runtime statistics to choose the most efficient query execution plan, one of the optimizations is automated conversion of shuffle join to a broadcast join.

`spark.sql.autoBroadcastJoinThreshold` - denotes the maximum size of a dataset that would be automatically broadcasted.
Spark SQL: JOIN Operations
Spark SQL: JOIN Operations

Broadcast Join

df_A

Join id | Col 3
---|---
D | faz_1
E | faz_1
A | faz_2
B | faz_2
C | faz_3

df_B

Join id | Col 1 | Col 2
---|---|---
A | foo_1 | bar_1
B | foo_1 | bar_1
C | foo_1 | bar_1
B | foo_2 | bar_2
C | foo_2 | bar_2
D | foo_2 | bar_2
C | foo_3 | bar_3
D | foo_3 | bar_3
E | foo_3 | bar_3

df_B.join(broadcast(df_A), ...)

df_A

Join id | Col 3
---|---
A | foo_1
B | foo_1
C | foo_1
D | foo_2
E | foo_2

df_B

Join id | Col 1 | Col 2
---|---|---
A | foo_1 | bar_1
B | foo_1 | bar_1
C | foo_1 | bar_1
B | foo_2 | bar_2
C | foo_2 | bar_2
D | foo_2 | bar_2
C | foo_3 | bar_3
D | foo_3 | bar_3
E | foo_3 | bar_3

Spark Driver

linkedin.com/in/aurimas-griciunas
Spark SQL: JOIN Operations (Data Skew case)

- Data skew is a condition in which a table’s data is unevenly distributed among partitions in the cluster.
- Data skew can severely downgrade performance of queries, especially those with joins.
- Joins between big tables require shuffling data and the skew can lead to an extreme imbalance of work in the cluster.
Spark SQL: JOIN Operations (Data Skew case)

JOIN Optimization parameters

`spark.sql.adaptive.optimizeSkewsInRebalancePartitions.enabled`
- When true and `spark.sql.adaptive.enabled` is true, Spark will optimize the skewed shuffle partitions in RebalancePartitions and split them to smaller ones

AQE mechanisms transparently discover and optimize implementation.
About data formats

Different workloads

• OLTP
  o Online transaction processing
  o Lots of small operations involving *whole rows*

• OLAP
  o Online analytical processing
  o Few large operations involving *subset of all columns*
The Parquet (hybrid) file format
Optimization: predicate pushdown

SELECT * FROM table WHERE \( x > 5 \)

Row-group 0: \( x \): \([\text{min: } 0, \text{ max: } 9]\)
Row-group 1: \( x \): \([\text{min: } 3, \text{ max: } 7]\)
Row-group 2: \( x \): \([\text{min: } 1, \text{ max: } 4]\)

...
Parquet: encoding schemes

- RLE_DICTIONARY

```
Uncompressed data
"United States"
"France"
"Germany"
"The Netherlands"
"The Netherlands"
"The Netherlands"
"France"
"United States"

Dictionary
0: "United States"
1: "France"
2: "Germany"
3: "The Netherlands"

Dictionary encoded data
0 1 2 3
3 3 1 0

Dictionary
0: "United States"
1: "France"
2: "Germany"
3: "The Netherlands"

Dictionary encoded data (RLE + bit-packed)
0 1 2 3,3 1 0
```
References

https://spark.apache.org/

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

https://www.agilelab.it/blog/spark-3-0-first-hands-on-approach-with-adaptive-query-execution-part-3