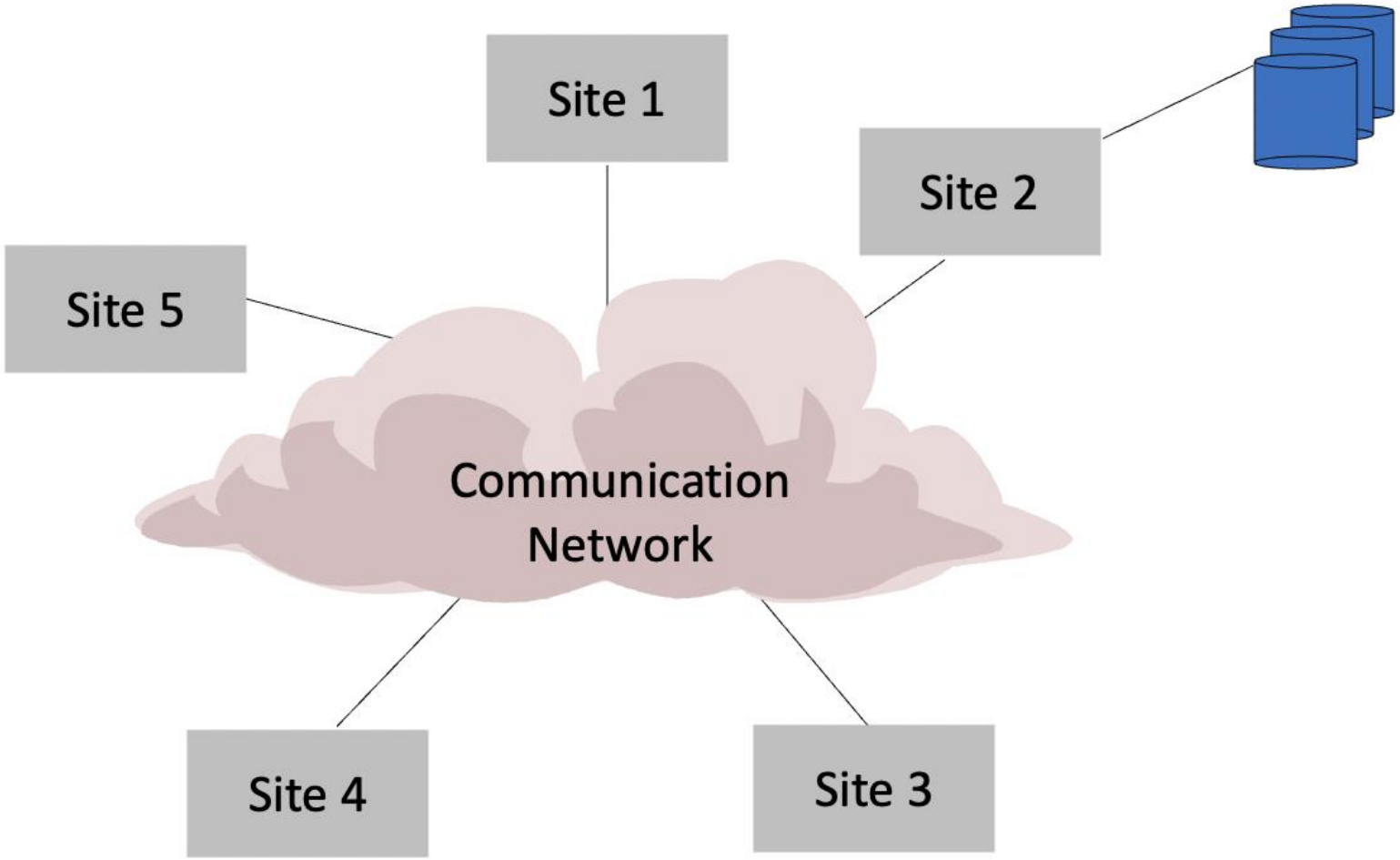
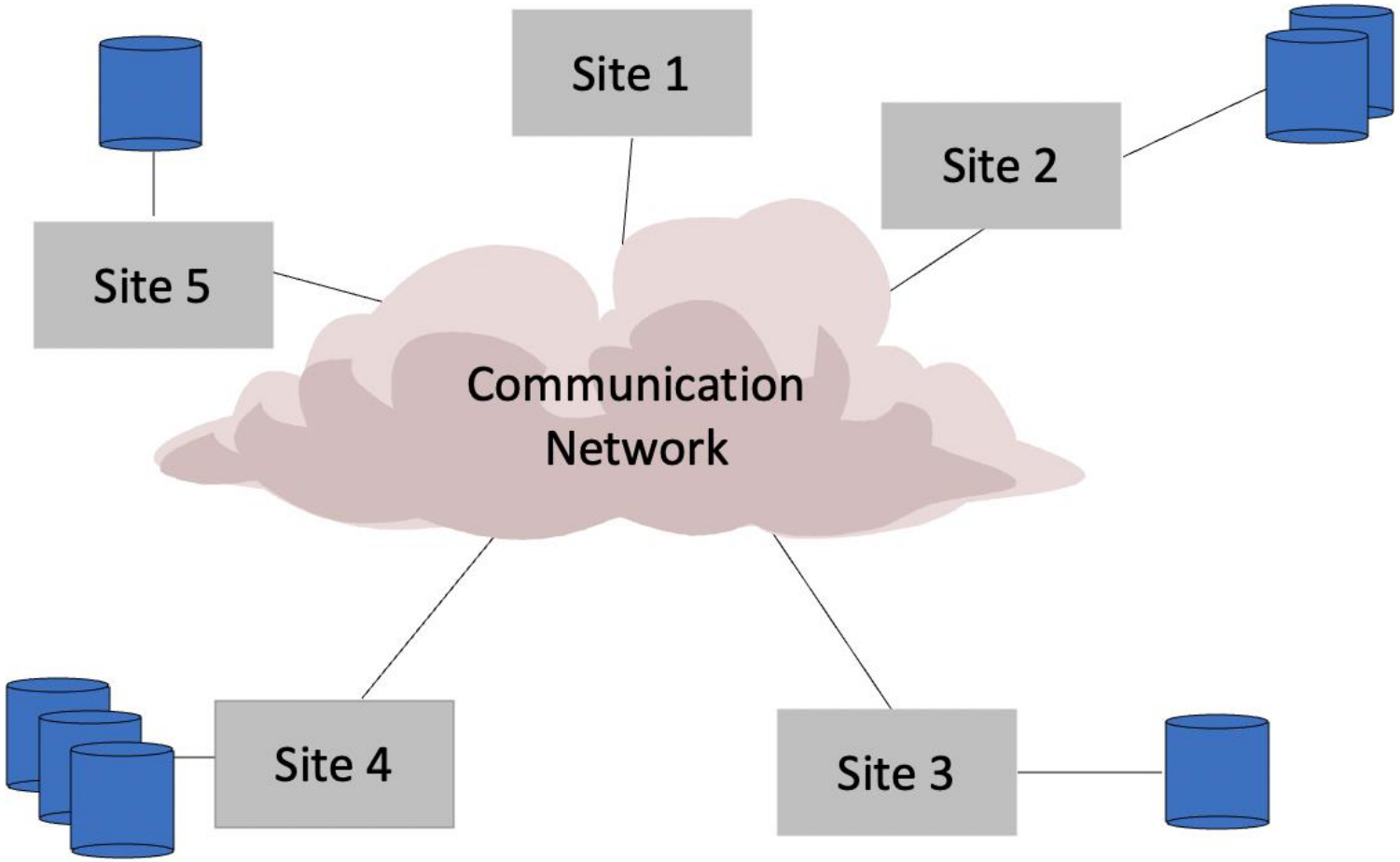
Cloud Based Data Engineering

Lecture 7: Spark

Centralized DBMS



Distributed DBMS



GFS

GFS – Google File System

Google File System (**GFS**) is a distributed file system developed by Google.

- A scalable distributed file system for large distributed data intensive applications
- Designed for system-to-system interaction, and **not for** user-to-system interaction.

The Google File System

GFS – Key Observations

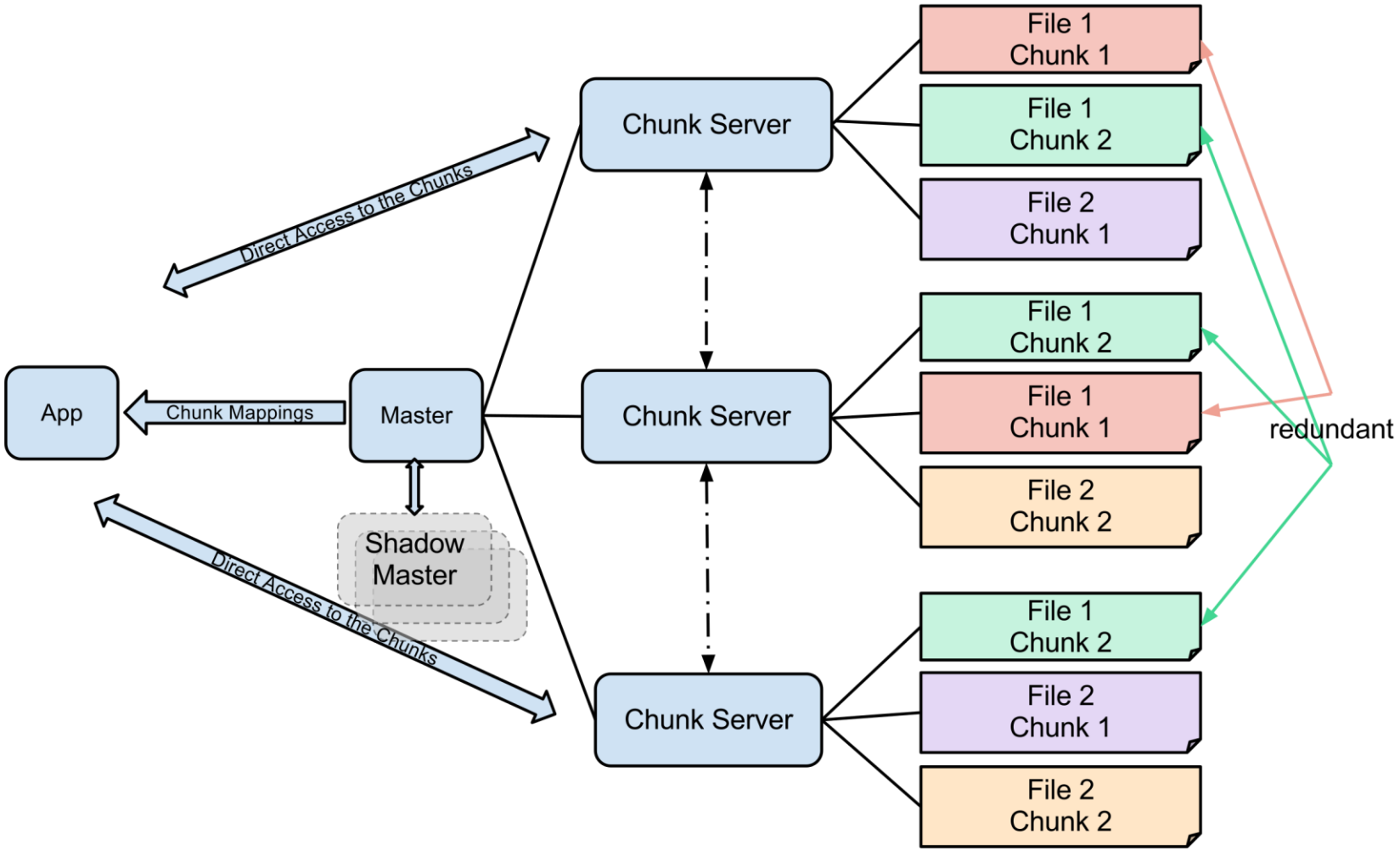
Component failures are the norm - constant monitoring, error detection, fault tolerance

Huge files - Multi GB files are common I/O operations

Files are mutated by appending new data - focus of performance optimization

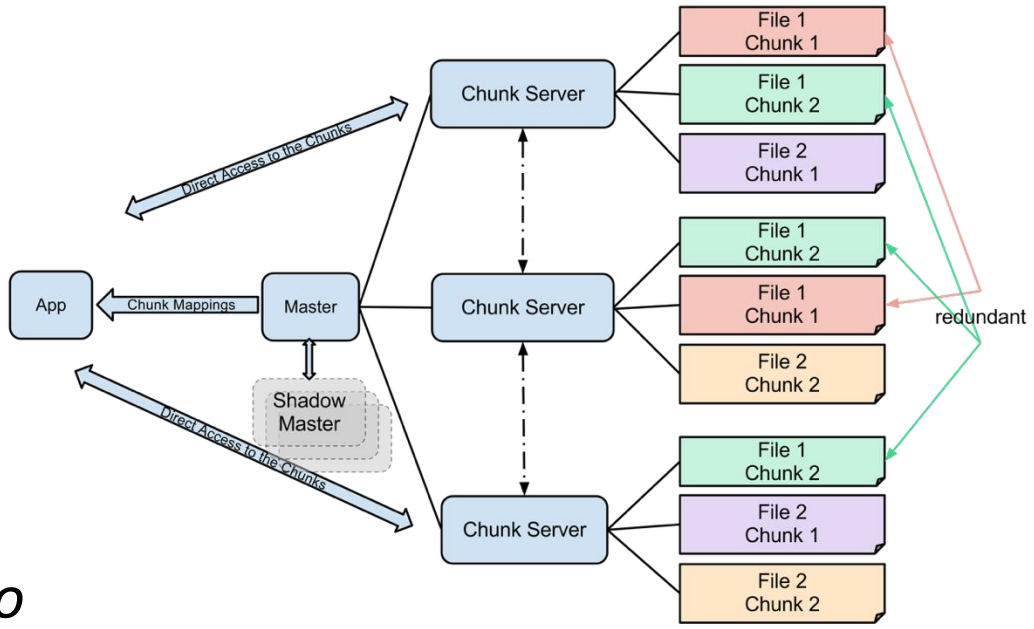
Co-designing the applications and APIs benefits overall system by increasing flexibility

GFS Architecture



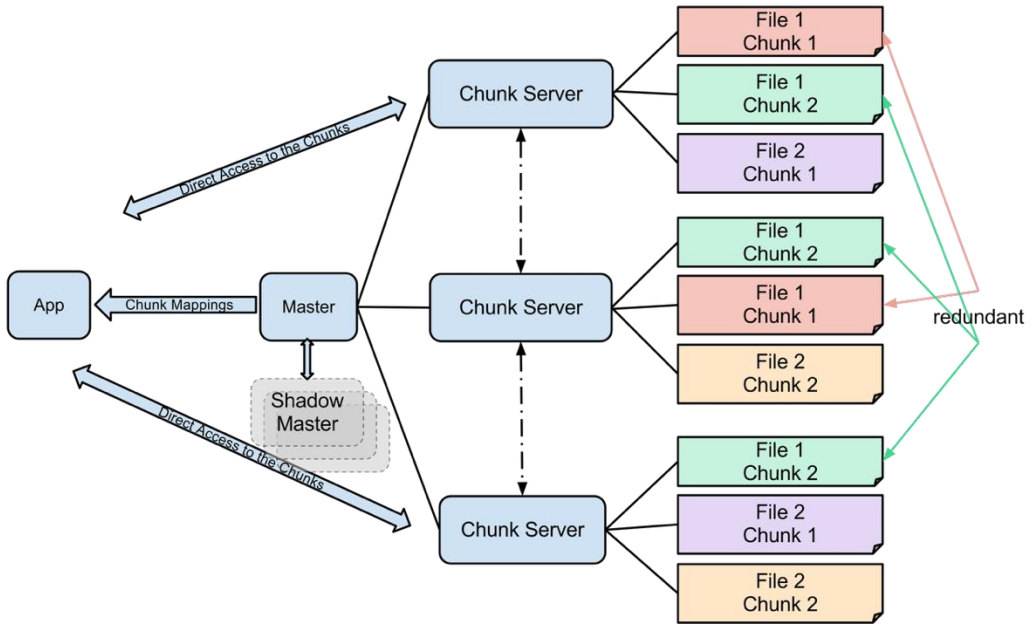
The Master

- Communicates with chunk-servers
- Give instructions
- Maintains all file system metadata
- Helps make sophisticated chunk placement and replication decision
- *For reading and writing, client contacts Master to get chunk locations, then deals directly with chunkservers*



Chunkservers

- Files are broken into chunks.
- Chunk size is 64 MB
- Each chunk is replicated on 3 (default) servers



Hadoop

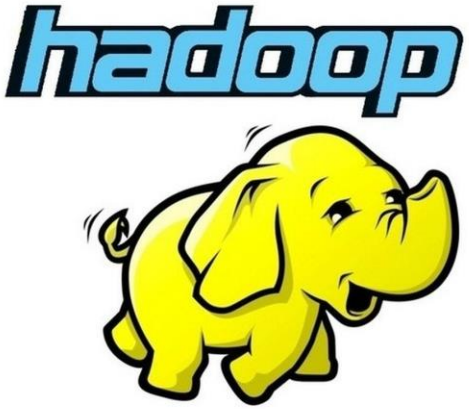
HADOOP

- **Hadoop** runs applications using the MapReduce algorithm,

  where the data is processed in parallel with others.

- **Hadoop** is used to develop applications that could perform complete statistical analysis

  on huge amounts of data.



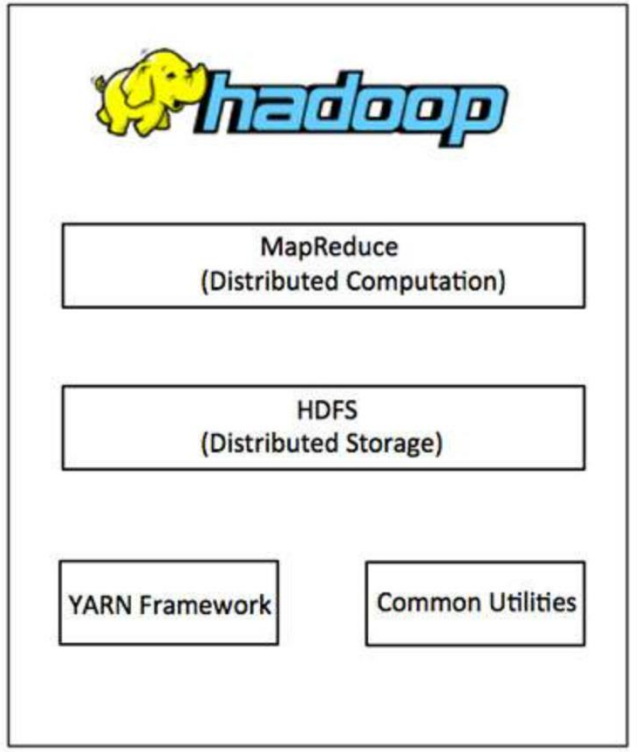
HADOOP

- Allows distributed processing of large datasets across clusters of computers
- Hadoop is designed to scale up from **single server** to **thousands of machines**
- It provides a distributed file system (**HDFS**) that stores data on the compute nodes, providing very high aggregate bandwidth across the cluster.

HADOOP Architecture

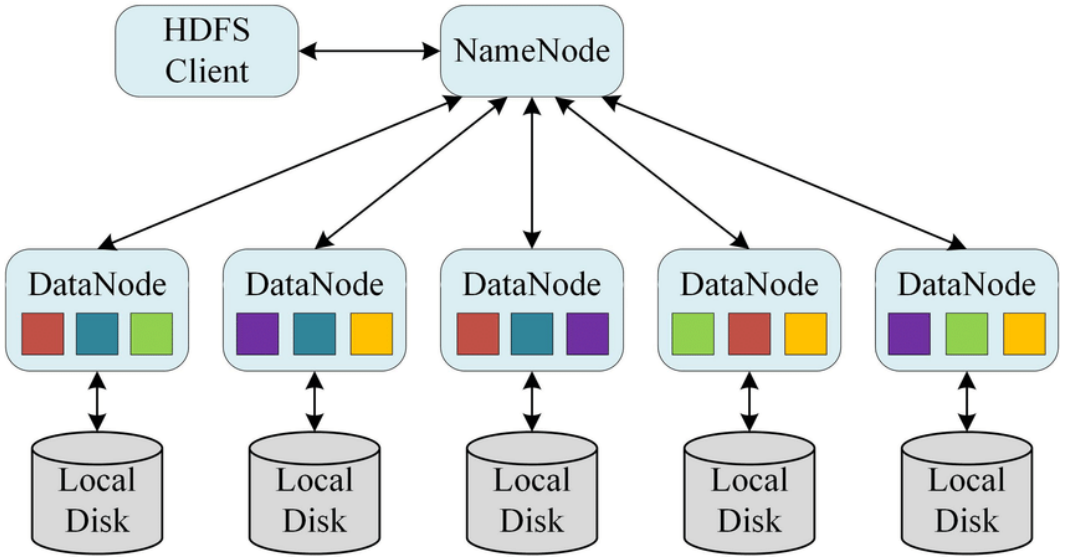
Hadoop has two major layers:

1. Processing/Computation layer (MapReduce)
2. Storage layer (Hadoop Distributed File System - HDFS)



HDFS

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on commodity hardware. It has many similarities with existing distributed file systems.



Spark

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

- Provides high-level APIs in Java, Scala, Python and R.
- Optimize engine that supports general execution graphs.
- Can manage "big data" collections with a small set of high-level primitives like map, filter, group-by, and join.
- Supports a rich set of higher-level tools including Spark SQL.

RDD - Resilient Distributed Datasets

- RDDs behave like Python collections (e.g. lists).

- Spark iteratively **apply functions** to every item of these collections in parallel to produce **new** RDDs.

- The data is distributed across nodes in a cluster of computers.

- Functions implemented in Spark can work in parallel across elements of the collection.

- RDDs automatically rebuilt on machine failure.

Operations in Distributed DBs

Two types of operations: **transformations** and **actions**

- Transformations are **lazy** (not computed immediately)

- Transformations are executed when an action is run

Transformations
map()
filter()
union()
distinct()
groupBy()
…

Actions
reduce()
collect()
count()
first()
saveAsTextFile()
..

Sample Spark transformations

map(func): passing each element of the source through func

filter(func): selecting those elements of the source on which returns true

union/intersection(otherDataset): contains the union/intersection of elements

join(otherDataset, [numTasks]): 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.

Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Operations in Distributed DBs - Transformation

map() flatMap()

filter()

mapPartitions() mapPartitionsWithIndex()

sample()

union() intersection() distinct()

groupBy() groupByKey()

reduceBy() reduceByKey()

sortBy() sortByKey()

join()

cogroup()

cartesian()

pipe()

coalesce()

repartition()

partitionBy()

Sample Spark Actions

reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

collect(): Return all the elements of the dataset as an array at the driver program.

This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

count(): Return the number of elements in the dataset.

Actions cause calculations to be performed
transformations just set things up (lazy evaluation)

Operations in Distributed DBs - Actions

reduce()

collect()

count()

first()

take()

takeSample()

saveToCassandra()

takeOrdered()

saveAsTextFile()

saveAsSequenceFile()

saveAsObjectFile()

countByKey()

foreach()

Spark – RDD Persistence

- You can persist (cache) an RDD
- When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)
- Allows future actions to be much faster (often >10x).
- Can choose storage level (MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, etc.)
- Can manually call unpersist()

RDD – Key Observations

- RDDs can be created from Hadoop InputFormats (such as HDFS files), “parallelize()” datasets, or by transforming other RDDs (you can stack RDDs)
- Actions can be applied to RDDs; actions force calculations and return values
- Lazy evaluation: Nothing computed until an action requires it
- RDDs are best suited for applications that apply the same operation **to all elements**

Life Cycle of Spark Program

1. Create input RDDs from external data or parallelize a collection in your driver program.

2. Lazily transform them to define new RDDs using transformations like `filter()` or `map()`

3. Ask Spark to cache() any intermediate RDDs that will need to be reused.

4. Launch actions such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark.

Spark DataFrames

Enable wider audiences beyond “Big Data” engineers to leverage the power of distributed processing.

- Inspired by data frames in R and Python (Pandas)
- Designed from the ground-up to support modern big data and data science applications
- Extension to the existing RDD API

DataFrames Are..

- The preferred abstraction in Spark
- Strongly typed collection of distributed elements
- Built on Resilient Distributed Datasets (RDD)
- Immutable once constructed

With DataFrames You Can..

With DataFrames you can:

- Track lineage information to efficiently recompute lost data
- Enable operations on collection of elements in parallel

construct DataFrames:

- by parallelizing existing collections (e.g., Pandas DataFrames)
- by transforming an existing DataFrames
- from files in HDFS or any other storage system (e.g., Parquet)

Apache Spark: Libraries “on top” of core that come with it

- Spark SQL
- Spark Streaming – stream processing of live datastreams
- MLlib - machine learning
- GraphX – graph manipulation
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge.

Gray sort competition: Winner

	Hadoop MR Record	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)