INDIAN INSTITUTE OF INFORMATION TECHNOLOGY

Distributed Computing Systems

Dr. Animesh Chaturvedi

Assistant Professor: IIIT Dharwad

Young Researcher: Heidelberg Laureate Forum

Postdoc: King's College London & The Alan Turing Institute

PhD: IIT Indore MTech: IIITDM Jabalpur

Distributed File System

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Network bisection bandwidth is limited
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

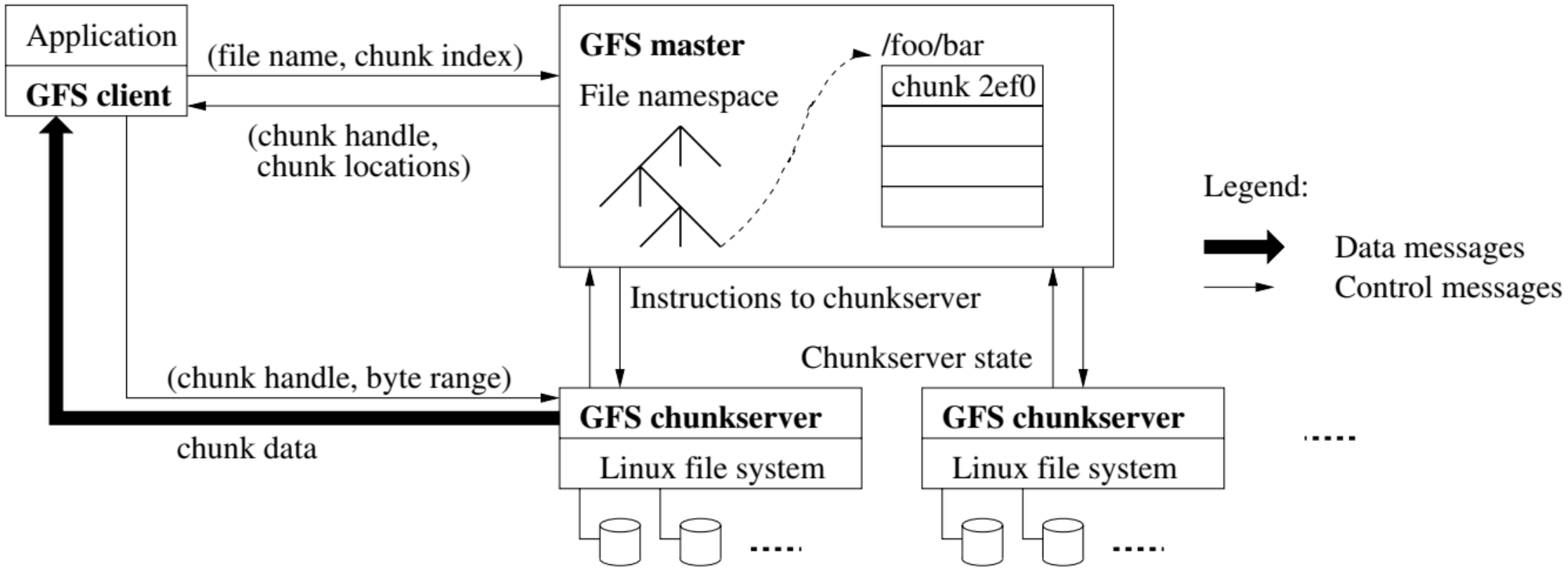
Big Files to Google File System (GFS)

- Earlier Google effort, "BigFiles", developed by Larry Page and Sergey Brin.
  - Supervisors: Hector Garcia-Molina, Rajeev Motwani, Jeff Ullman, and Terry Winograd
- “Big File” was regenerated as “Google File System” by Sanjay Ghemawat, et al.
- Google File System (GFS)
  - “It is widely deployed within Google as the storage platform for the generation and processing of data used by Google service as well as research and development efforts that require large data sets.” 2003
  - “The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.” 2003

http://infolab.stanford.edu/~backrub/google.html

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

Google File System (GFS)



https://en.wikipedia.org/wiki/Google_File_System

https://sites.google.com/site/gfsassignmentwiki/home

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

GFS: Assumptions

- Choose commodity hardware over "exotic" hardware
  - Scale "out", not "up"
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

https://en.wikipedia.org/wiki/Google_File_System
https://sites.google.com/site/gfsassignmentwiki/home
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas implemented in Java)

GFS to HDFS

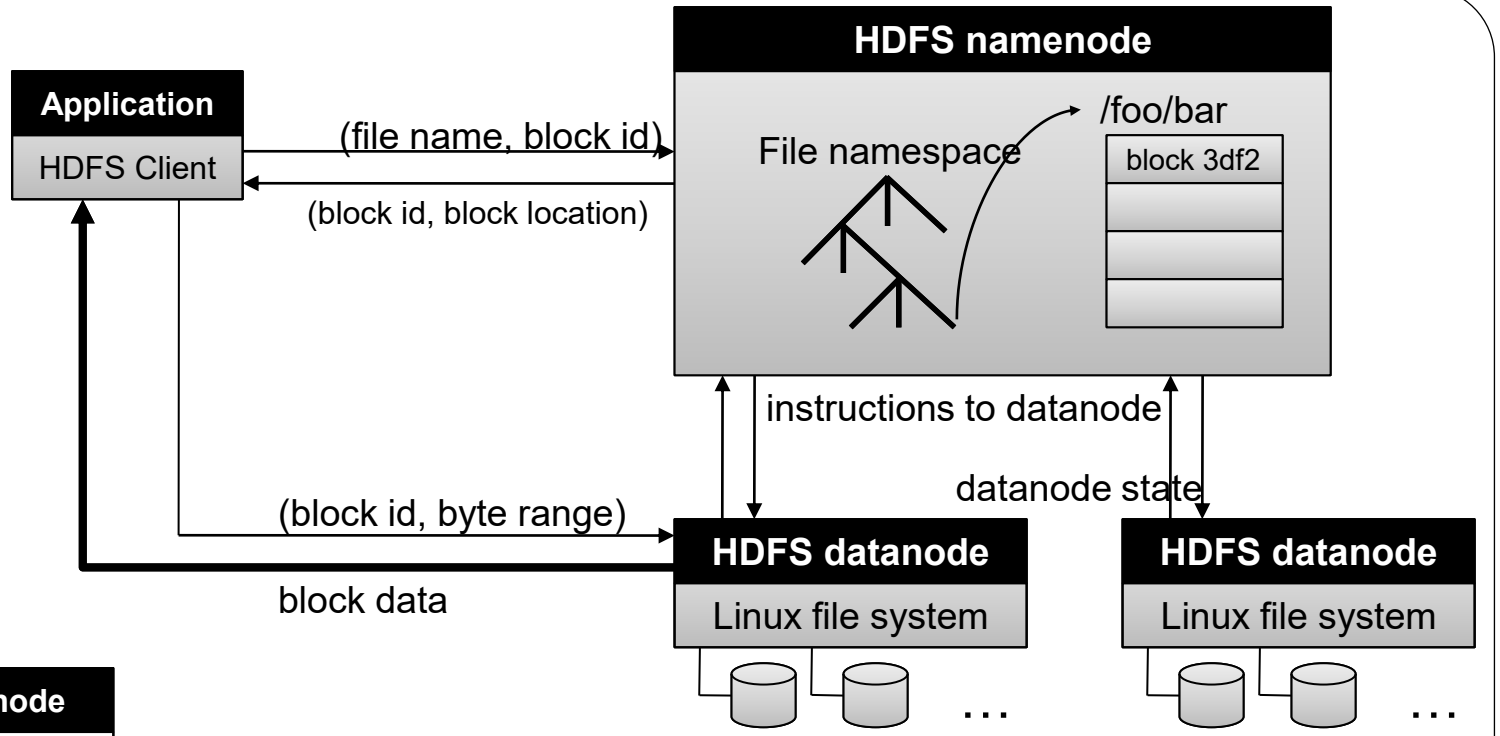
- Google File System (GFS) has similar open-source
  - "Hadoop Distributed File System (HDFS)"
- GFS and HDFS are distributed computing environment to process "Big Data".
- GFS and HDFS are not implemented in the kernel of an operating system, but they are instead provided as a userspace library.
- GFS and HDFS properties
  - Files are divided into fixed-size chunks of 64 megabytes.
  - Scalable distributed file system for large distributed data intensive applications.
  - Provides fault tolerance.
  - High aggregate performance to a large number of clients.

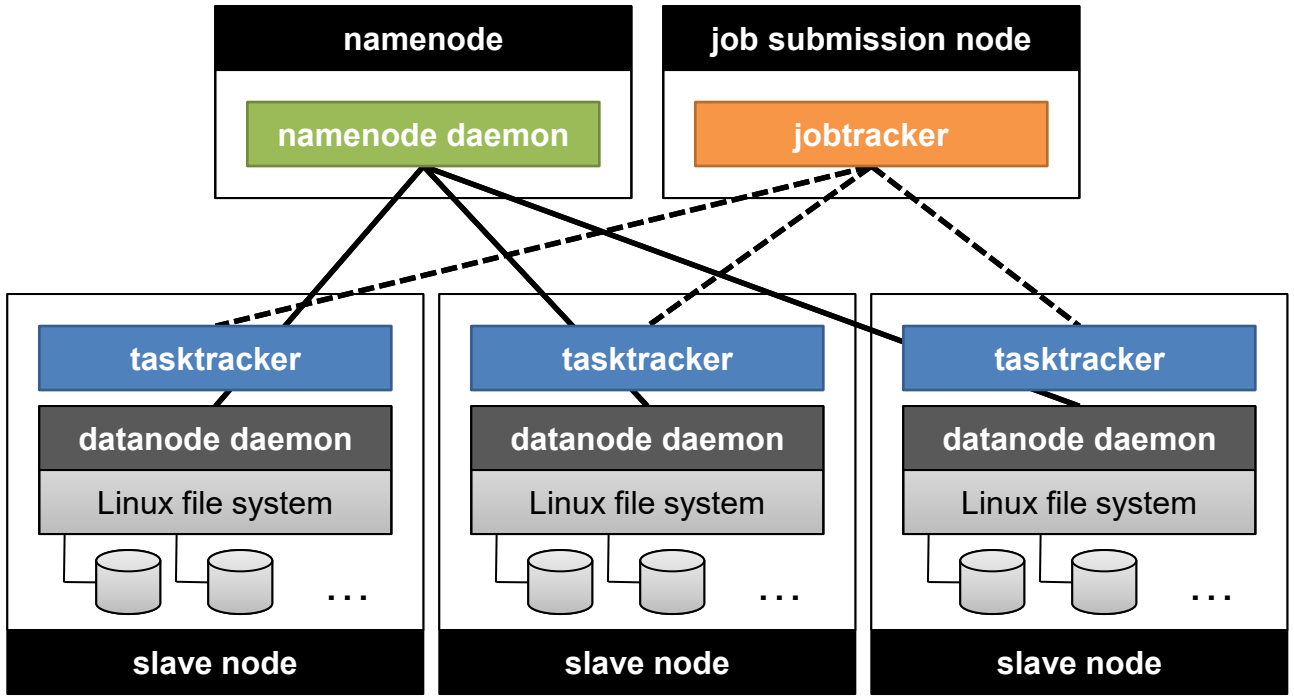
https://en.wikipedia.org/wiki/Google_File_System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.

GFS to HDFS

- GFS master
  - Hadoop namenode
- GFS chunkservers
  - Hadoop datanodes

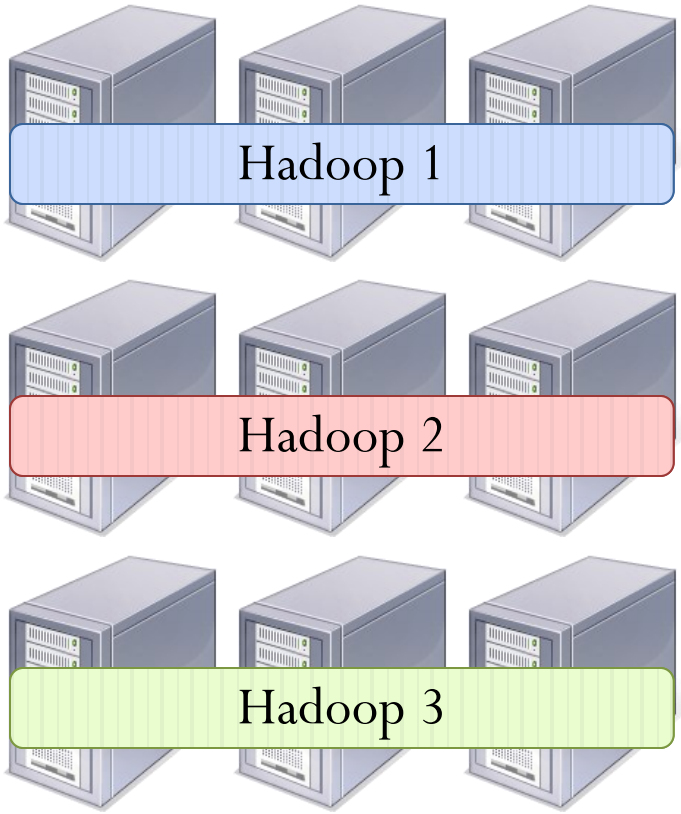




Coarse-grained sharing

Option: Coarse-grained sharing

- Give framework a (slice of) machine for its entire duration



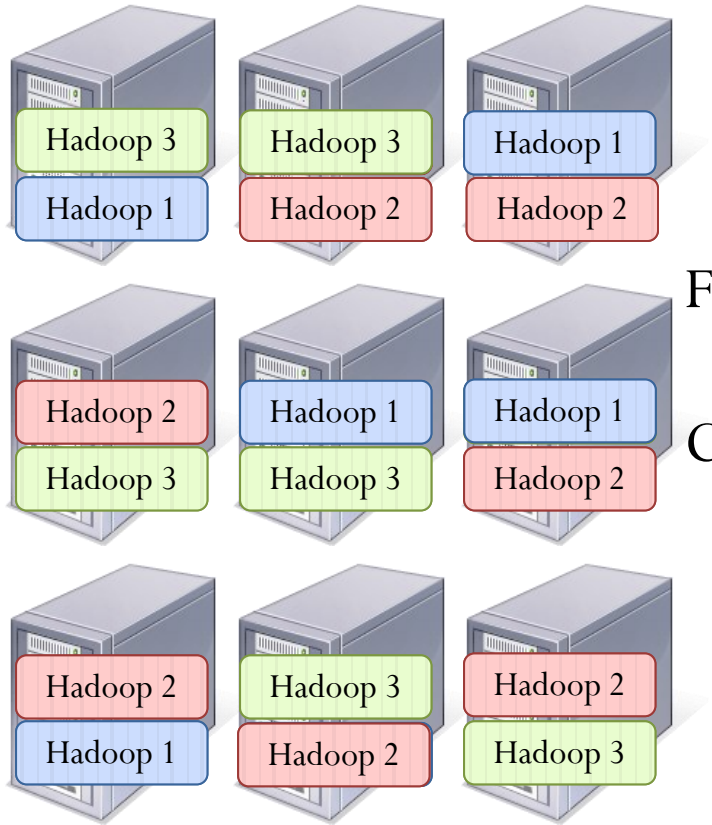
Data locality compromised if machine held for long time

Hard to account for new frameworks and changing demands

→ **hurts utilization and interactivity**

Fine-grained sharing

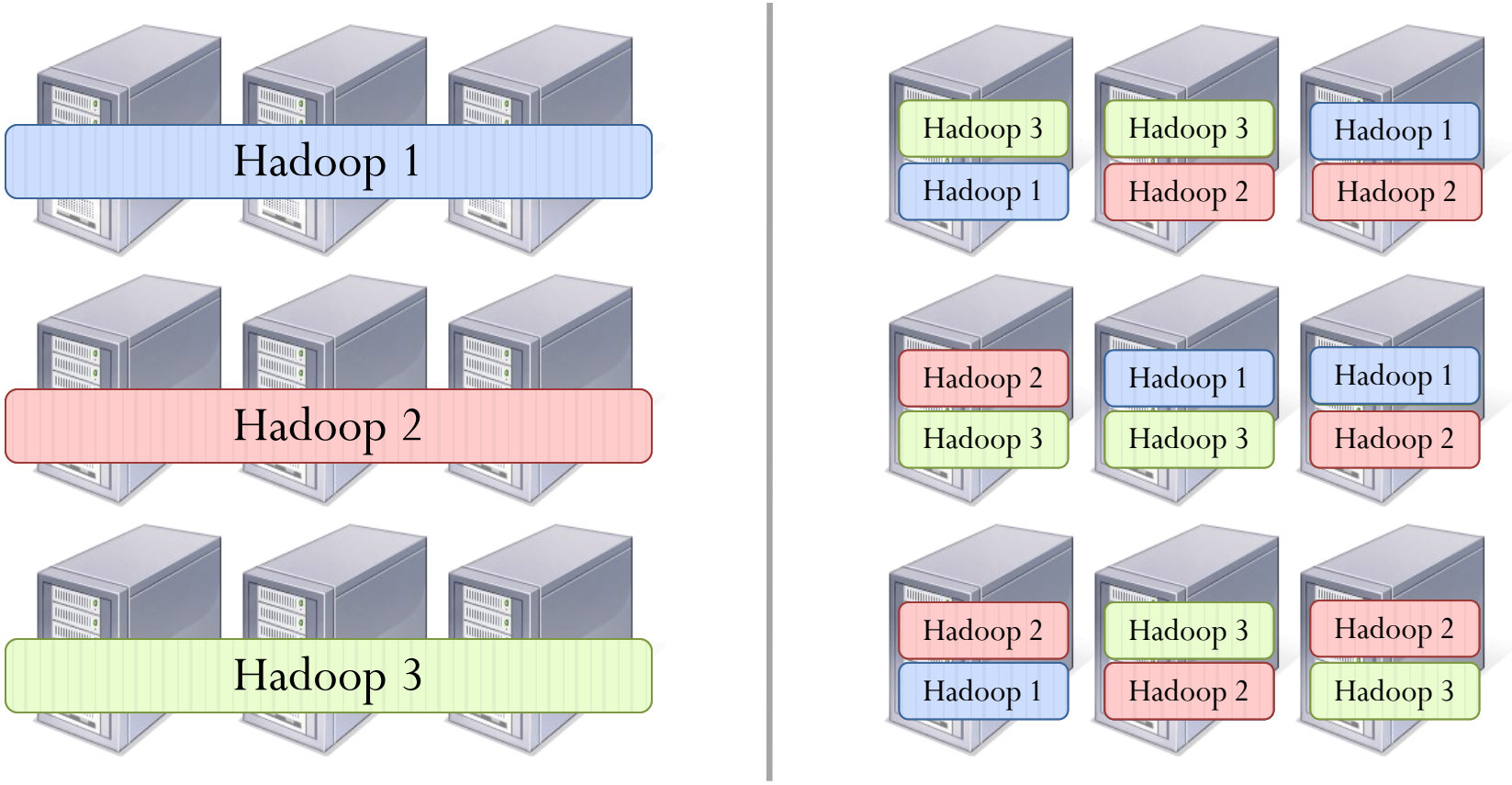
- Support frameworks that use smaller *tasks* (in time and space) by multiplexing them across all available resources



Frameworks can take turns accessing data on each node

Can resize frameworks shares to get utilization & interactivity

Multiple Hadoops Experiment



Big Data

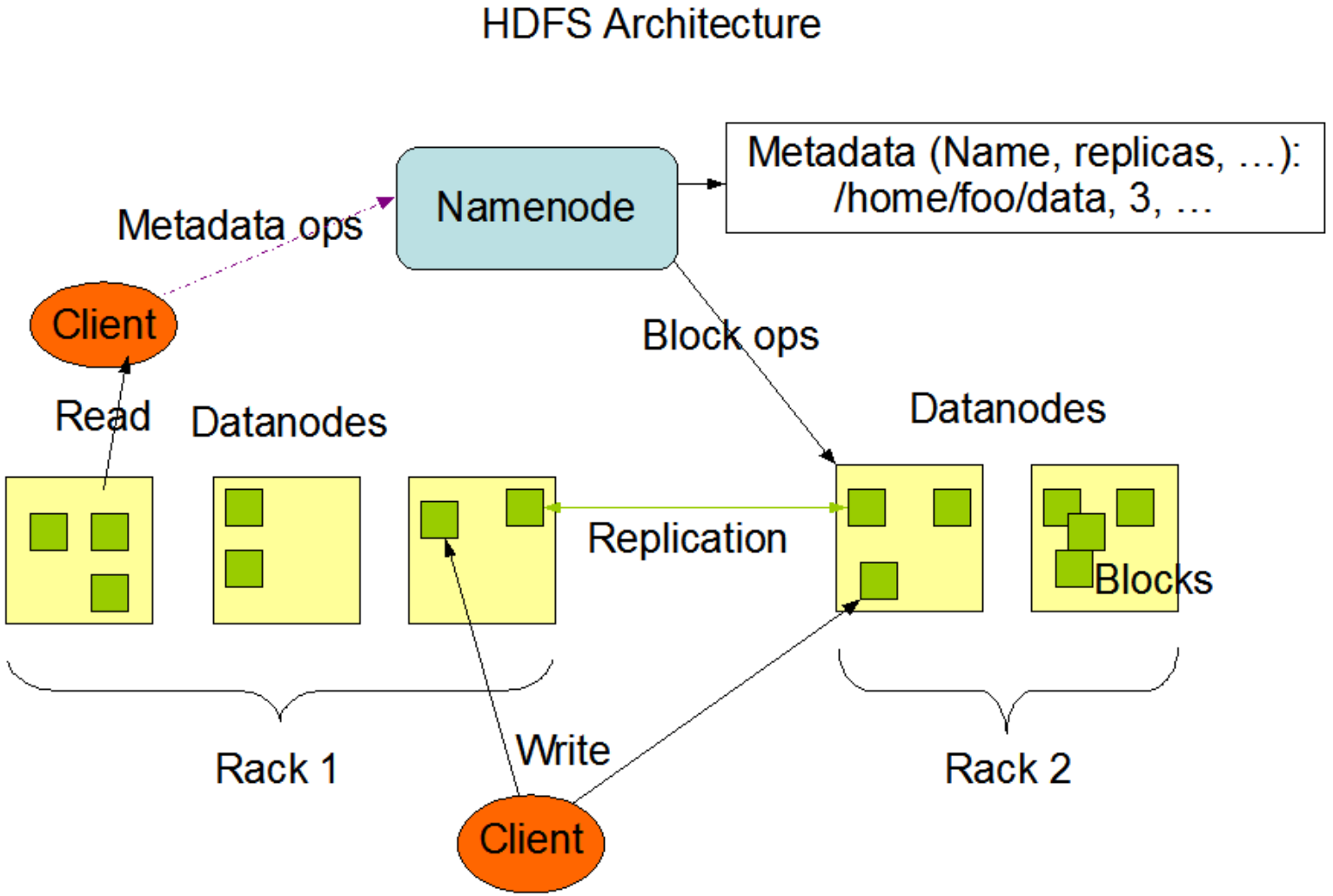
- Big data can be described by the following characteristics:
  - Volume: size large than terabytes and petabytes
  - Variety: type and nature, structured, semi-structured or unstructured
  - Velocity: speed of generation and processing to meet the demands
  - Veracity: the data quality and the data value
  - Value: Useful or not useful
- The main components and ecosystem of Big Data
  - Data Analytics: data mining, machine learning and natural language processing etc.
  - Technologies: Business Intelligence, Cloud computing & Databases etc.
  - Visualization: Charts, Graphs etc.

Hadoop

- “The Apache Hadoop project develops open source software for reliable, scalable, distributed computing.” Software library and a framework.
- Created by Doug Cutting Named on his son's stuffed elephant
- For distributed processing of large data sets across clusters of computers using simple programming models.
- Locality of reference
- Scalability: Scale up from single servers to thousands of machines,
  - Each offering local computation and storage
  - Program remains same for 10, 100, 1000,… nodes
  - Corresponding performance improvement
- Fault-tolerant file system:
  - Detect and handle failures and Delivering a highly-available.
- Hadoop Distributed File System (HDFS) Modeled on Google File system
- MapReduce for Parallel computation using
- Components – Pig, Hbase, HIVE, ZooKeeper

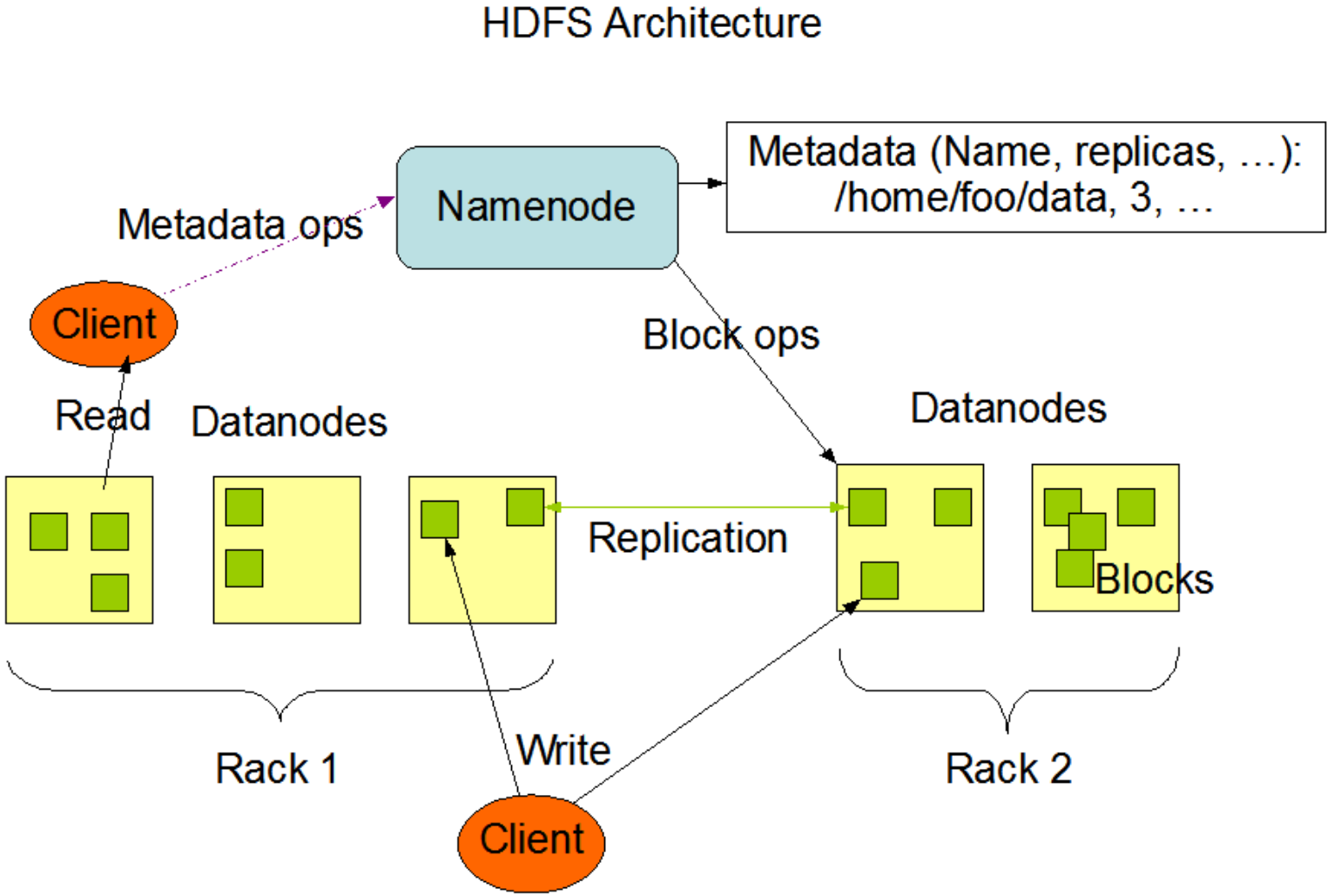
Hadoop Distributed File System (HDFS)

- HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients.
- In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.
- HDFS exposes a file system namespace and allows user data to be stored in files.



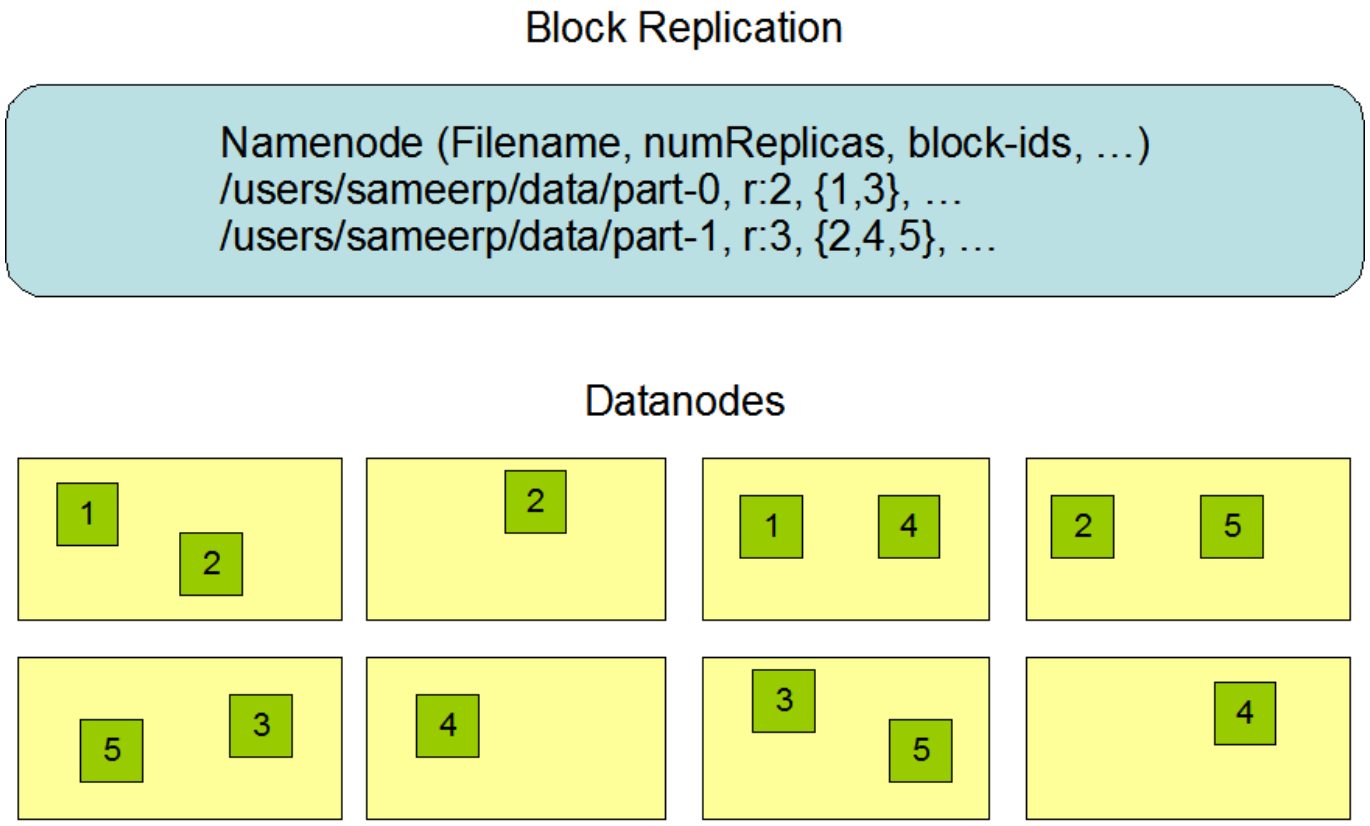
Hadoop Distributed File System (HDFS)

- Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.
- The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.
- The DataNodes are responsible for serving read and write requests from the file system's clients.
- The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode



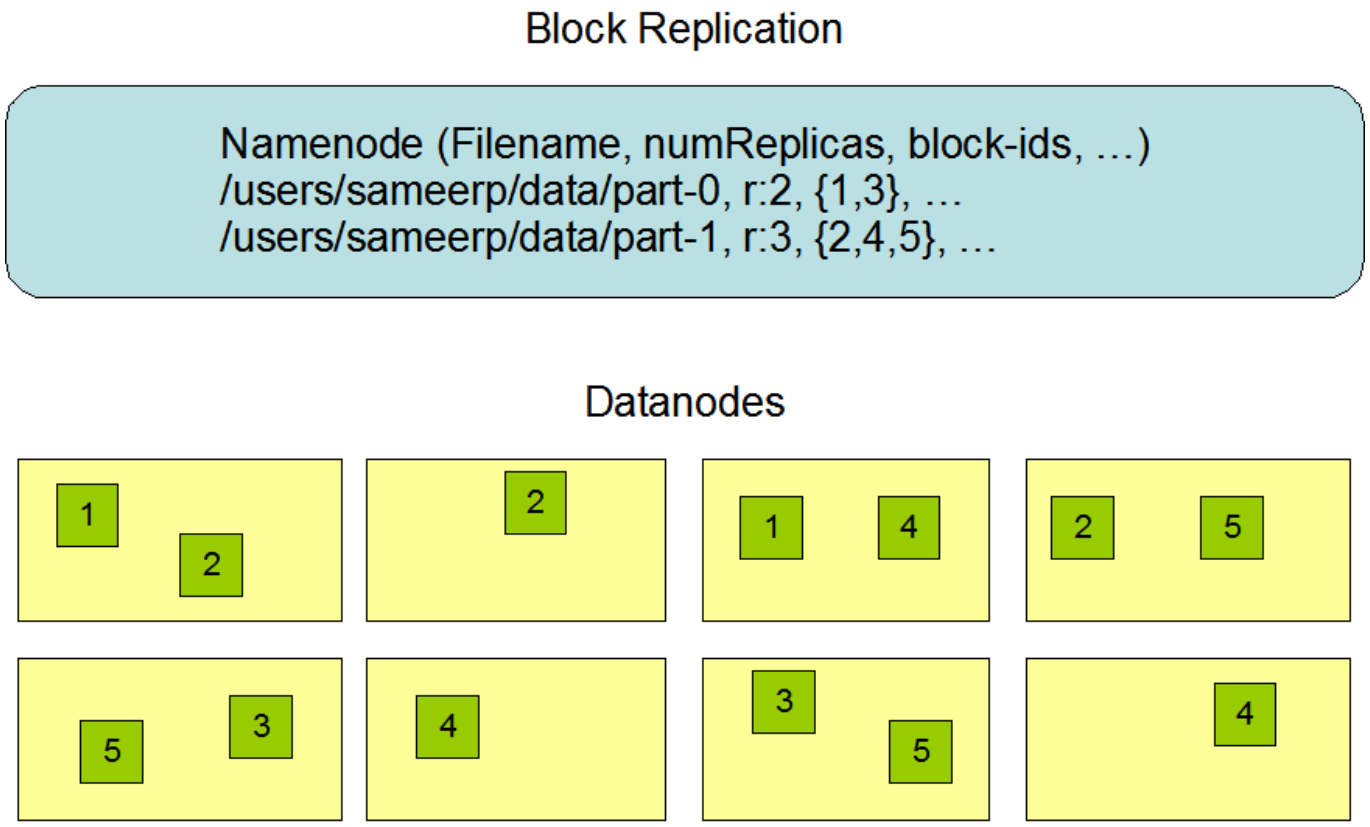
Namenode Responsibilities

- Manages File System namespace
  - mapping files to blocks and blocks to data nodes.
  - Maintains status of data nodes
  - holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Maintaining overall health:
  - Periodic communication with the Datanodes
  - Block re-replication and rebalancing
  - Garbage collection
- Heartbeat: Datanode sends heartbeat at regular intervals, if heartbeat is not received, Datanode is declared to be dead
- Coordinating file operations:
  - Directs clients to Datanodes for reads and writes
  - No data is moved through the Namenode
- Blockreport: Datanode sends list of blocks on it. Used to check health of HDFS

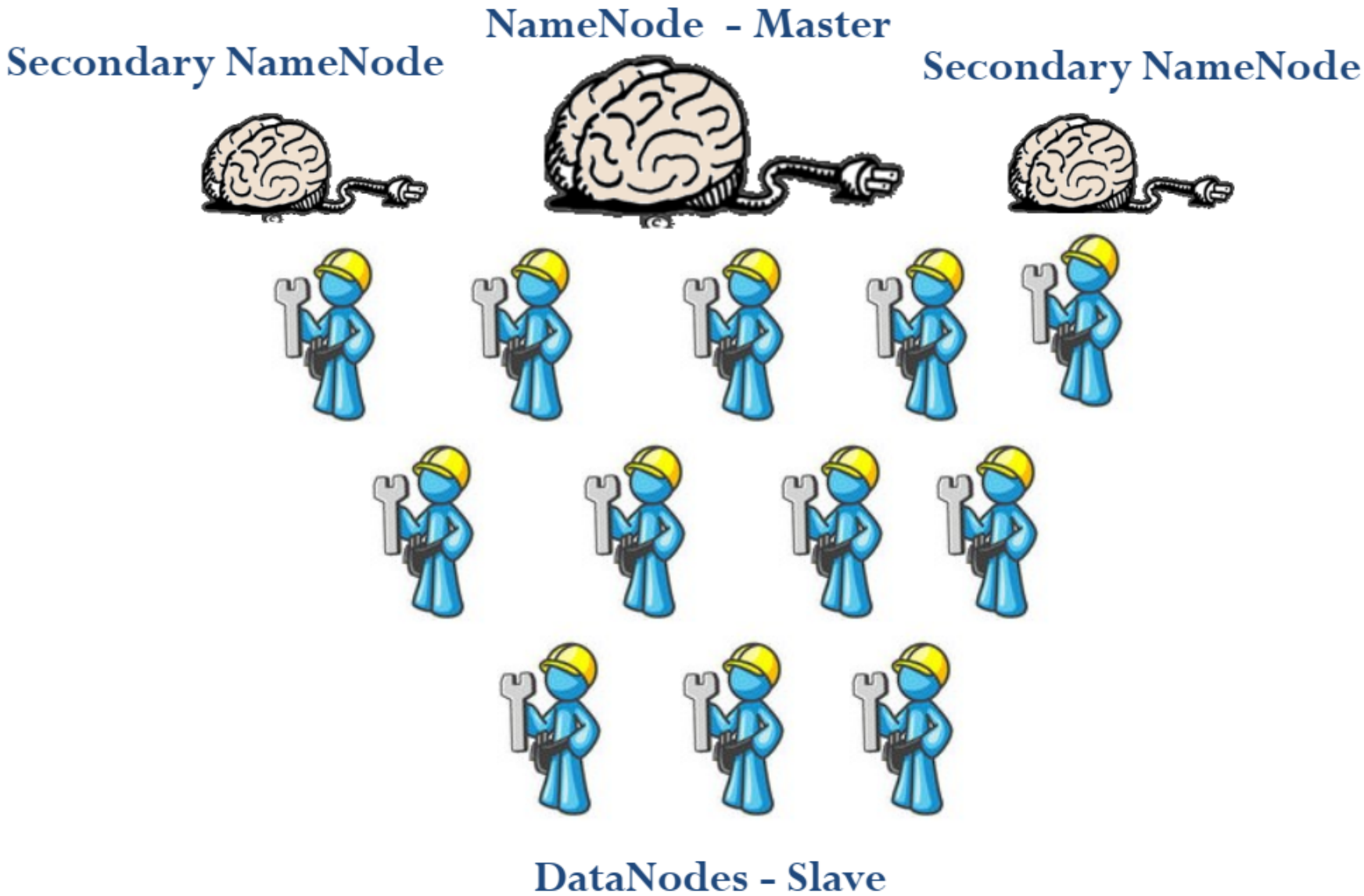


Datanodes: Responsibilities

- Replicates
  - On Datanode failure,
  - On Disk failure,
  - On Block corruption
- Data integrity
  - Checksum for each block,
  - Stored in hidden file
- Rebalancing - balancer tool
  - Provisioning: addition of new nodes,
  - Decommissioning: remove node,
  - Deletion of some files



HDFS Force


NameNode - Master
Secondary NameNode
Secondary NameNode
DataNodes - Slave

Map Reduce

MapReduce is a programming model

- An implementation for processing and generating large data sets.
- Many real world tasks are expressible in this model.
- Users specify
  - a map function that processes a key/value pair to generate a set of intermediate key/value pairs,
  - a reduce function that merges all intermediate values associated with the same intermediate key.
- MapReduce computation processes
  - many terabytes of data
  - on thousands of machines.
- Runs on a large cluster of commodity machines and is highly scalable.

Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

MapReduce for programmer

- Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines.
- Programmers
  - find it easy to use,
  - easily utilize the resources of a large distributed system,
  - without any experience with parallel and distributed systems
- Hundreds of MapReduce programs have been implemented
- Thousands of MapReduce jobs are executed on Google's clusters every day.
- Takes care of
  - partitioning the input data,
  - scheduling the program's execution across a set of machines,
  - handling machine failures, and
  - managing the required inter-machine communication.

Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

Typical Large-Data Problem

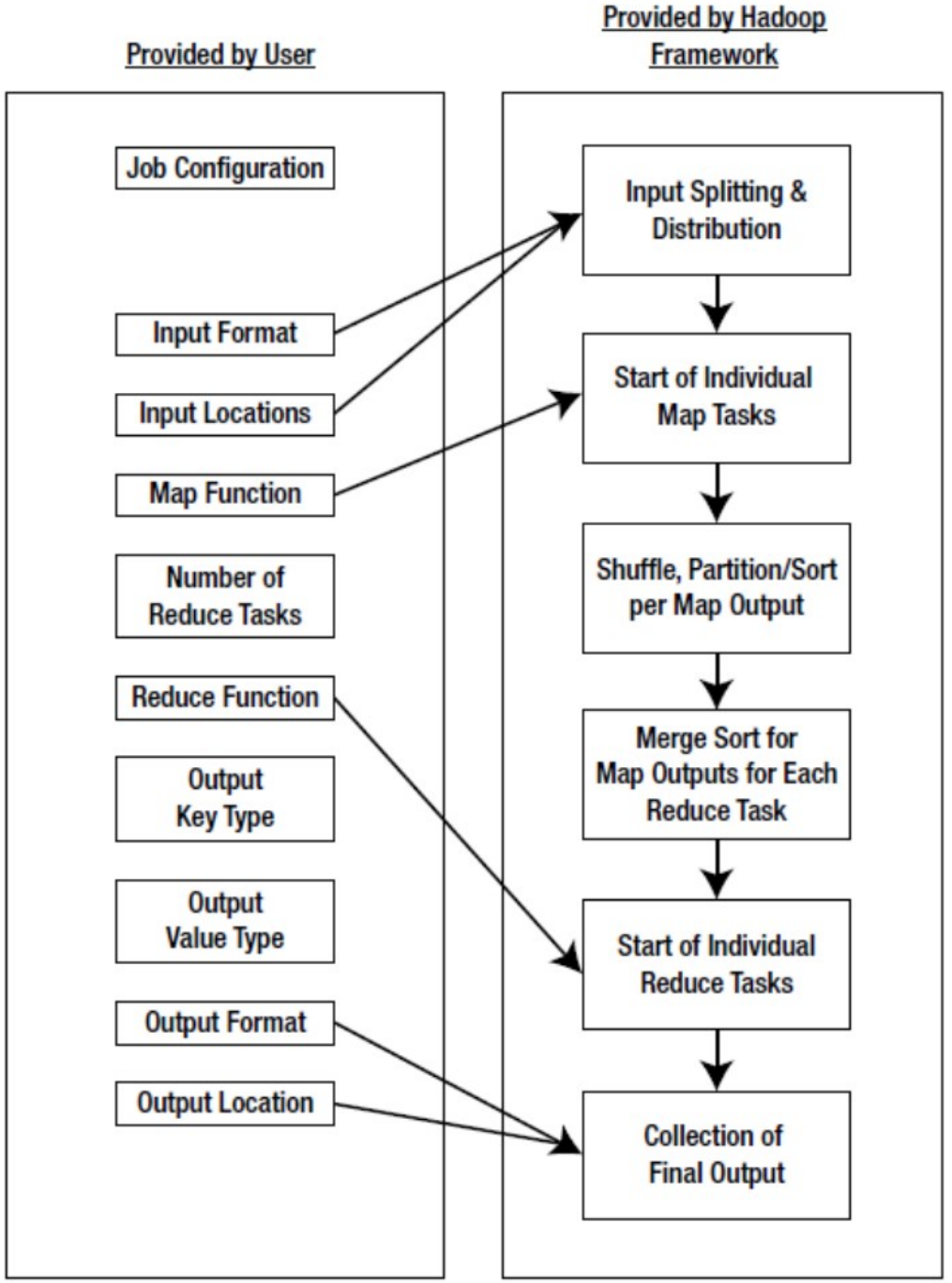
- Iterate over a large number of records **Map**
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results **Reduce**
- Generate final output

Key idea: provide a functional abstraction for these two operations – MapReduce

Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

Map Reduce

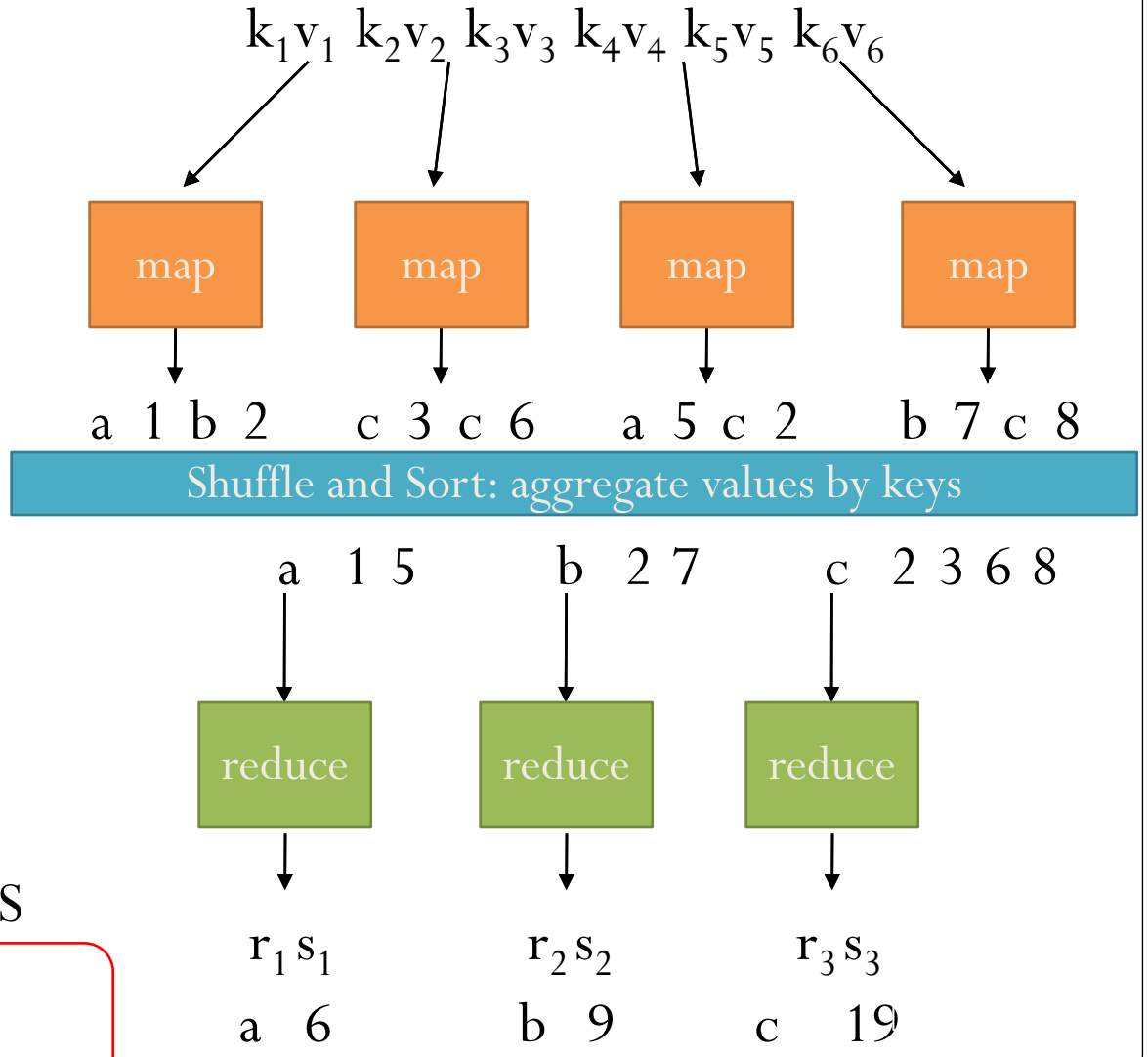
- Format of input- output (key, value)
  - Map: (k1, v1) → list (k2, v2)
  - Reduce: (k2, list v2) → list (k3, v3)
  - All values with the same key are sent to the same reducer
- The execution framework handles everything else
- MapReduce will not work for
  - Inter-process communication
  - Data sharing required
  - Example: Recursive functions



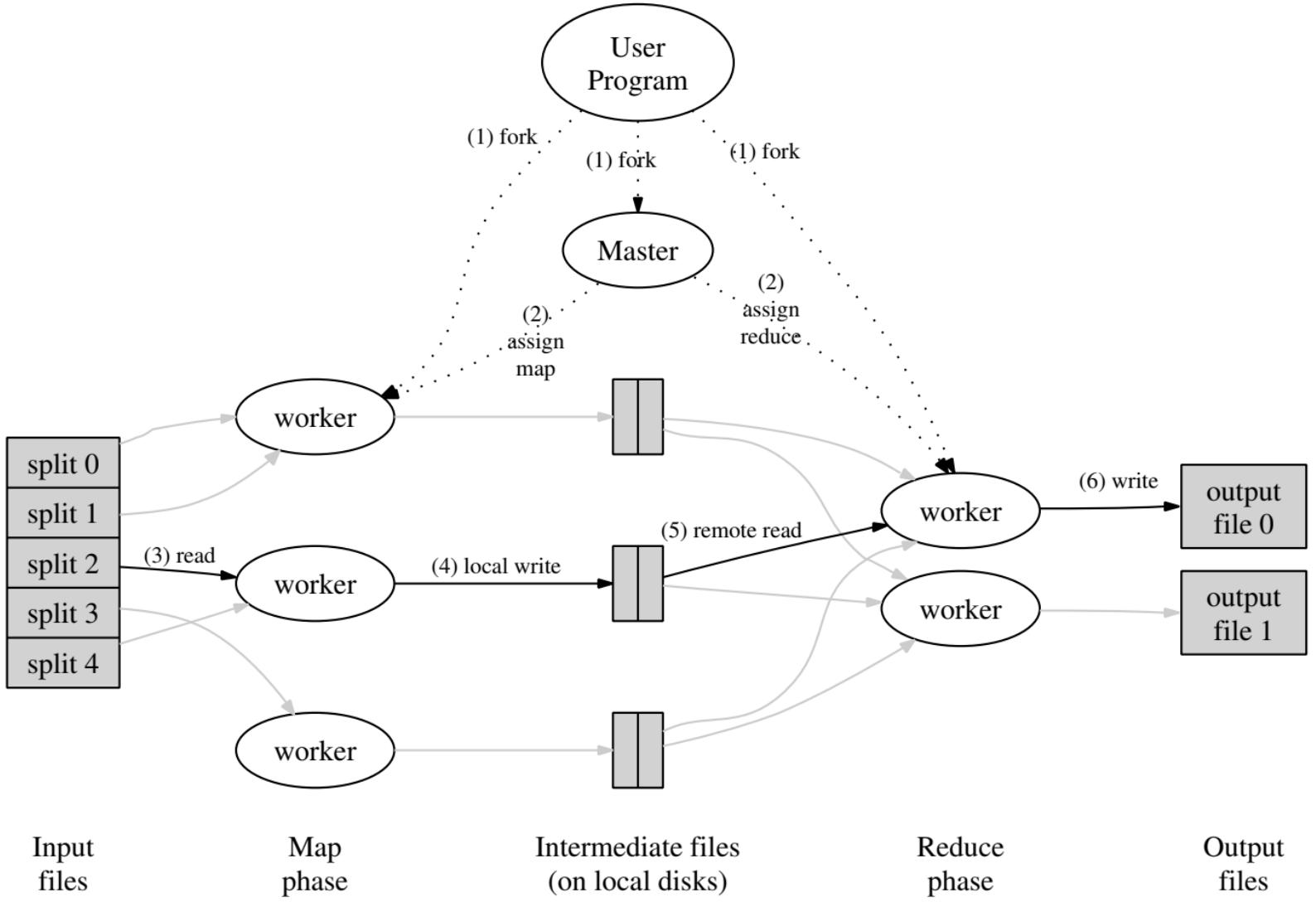
MapReduce Runtime

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and automatically restarts
- Handles speculative execution
  - Detects "slow" workers and re-executes work
- Everything happens on top of a Distributed FS

Sounds simple, but many challenges!



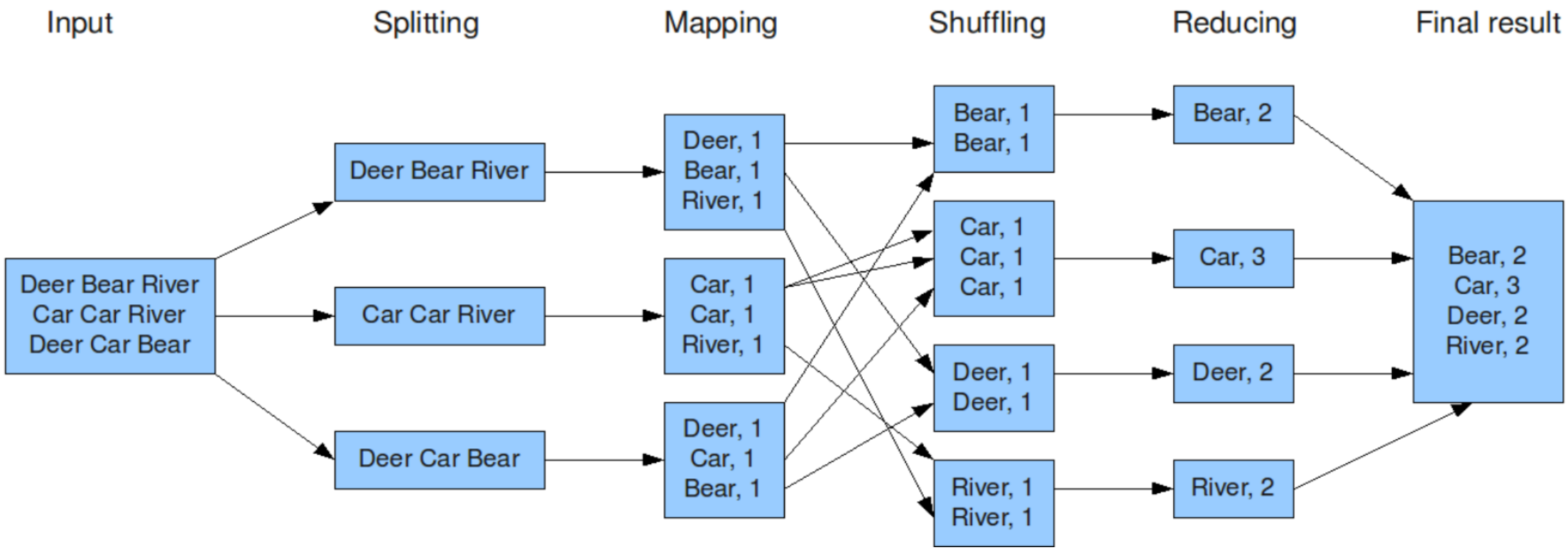
MapReduce Overall Architecture



Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

Word Count

- **Map:** Input lines of text to breaks them into words gives outputs for each word <key = word, value =1 >
- **Reduce:** Input <word, 1> output <word, + value>



“Hello World” Example: Word Count

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem, but still lots of room for improvement
- Lots of custom research implementations issues

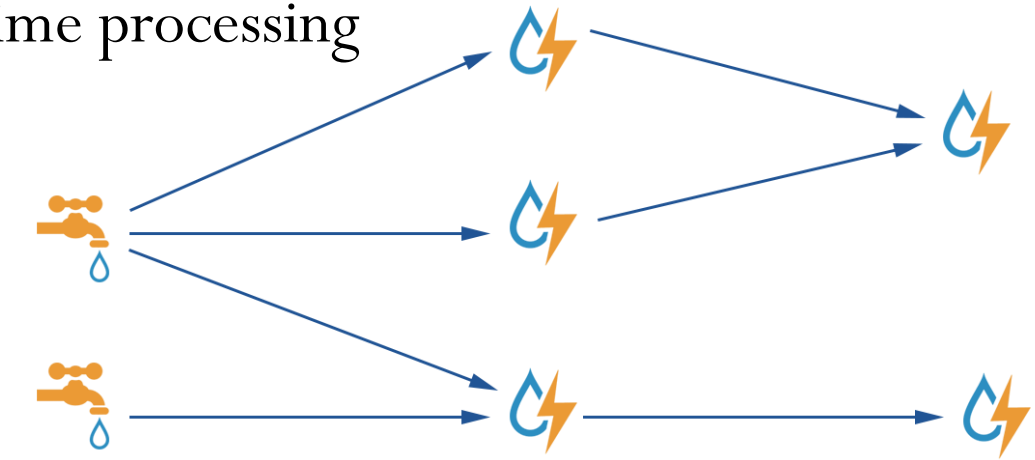
Map Reduce/GFS Summary

- Simple, but powerful programming model
- Scales to handle petabyte+ workloads
  - Google: six hours and two minutes to sort 1PB (10 trillion 100-byte records) on 4,000 computers
  - Yahoo!: 16.25 hours to sort 1PB on 3,800 computers
- Incremental performance improvement with more nodes
- Seamlessly handles failures, but possibly with performance penalties

Storm

Storm

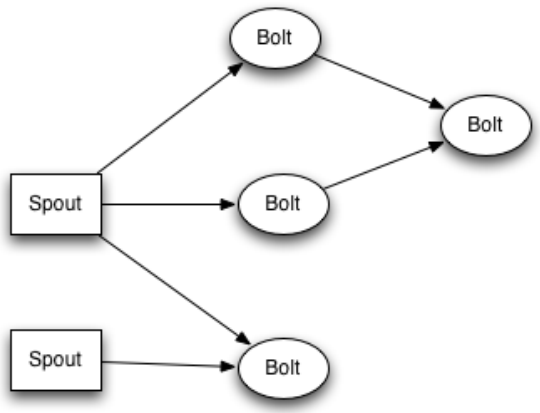
- Reliably for processing unbounded streams of data
- Hadoop for batch processing. Storm real-time processing
- Realtime analytics
- Online machine learning
- Continuous computation
- Distributed RPC.

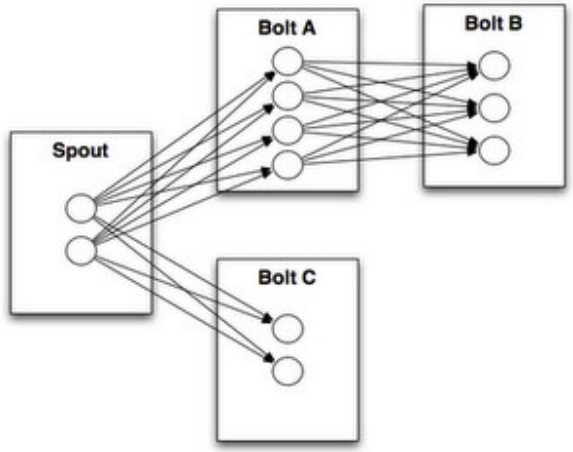


- A million tupples can be processed per second per node.
- It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Storm

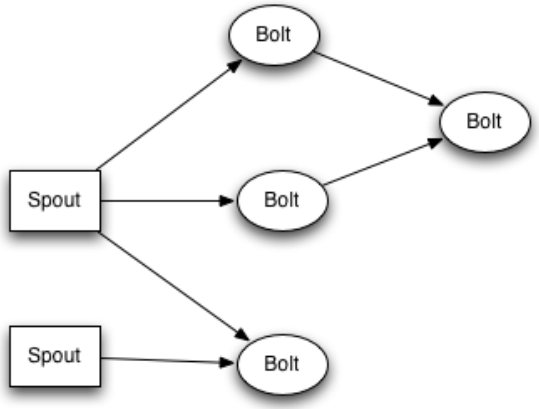
- Apache Storm is a free and open source distributed Realtime computation system.
- Apache Storm makes it easy to reliably process unbounded streams of data, doing for Realtime processing what Hadoop did for batch processing.
- Apache Storm integrates with the queueing and database technologies you already use.
- An Apache Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed.





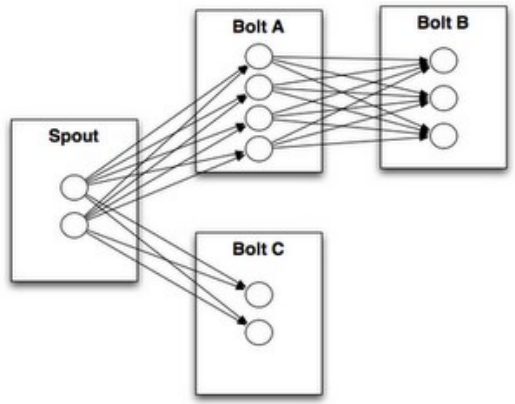
Storm

- Topologies: analogous to a MapReduce job. MapReduce job finishes, whereas a topology runs forever or until you kill it.
  - A topology is a graph of spouts and bolts that are connected with stream groupings.
- Streams: is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion.
  - Streams are defined with a schema that names the fields in the stream's tuples.
  - Schema are integers, longs, shorts, bytes, strings, doubles, floats, booleans, and byte arrays.
  - Every stream is given an id when declared.



Storm

- **Spouts:** A spout is a source of streams in a topology. Generally spouts will read tuples from an external source and emit them into the topology.
  - **Reliable Spouts:** Replaying a tuple if it failed to be processed.
  - **Unreliable Spouts:** Forgets about the tuple as soon as it is emitted.

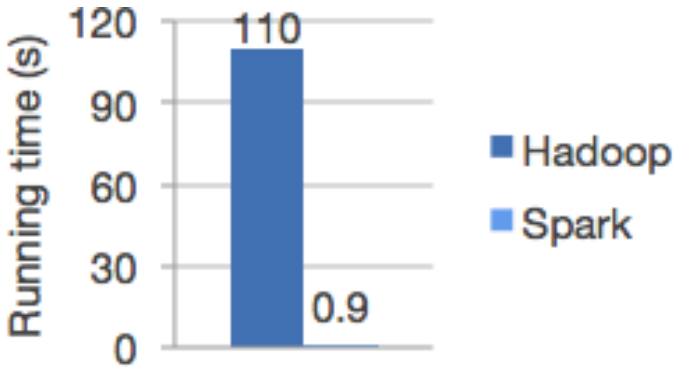


- **Bolts:** All processing in topologies is done in bolts.
  - Bolts can do filtering, functions, aggregations, joins, talking to databases, and more.
  - Bolts can do stream transformations into a new stream in a distributed and reliable way.
  - Complex stream transformations often requires multiple steps and thus multiple bolts.
  - For example, transform a stream of tweets into a stream of trending topics.

Spark

Apache Spark

- Unified analytics engine for large-scale data processing.
- Speed: Run workloads 100x faster.
- Both batch and streaming data, using Directed Acyclic Graph (DAG) scheduler, a query optimizer, and a physical execution engine.
- Ease of Use: Write applications quickly in Java, Scala, Python, R, and SQL.
- Spark offers 80+ high-level operators to build parallel apps.



https://spark.apache.org/

Apache Spark

- Combine SQL, streaming, and complex analytics. Spark libraries
  - SQL and DataFrames,
  - MLlib for machine learning,
  - GraphX, and
  - Spark Streaming.
- Spark runs on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud.
- Run Spark using its standalone cluster mode, on EC2, on Hadoop YARN, on Mesos, or on Kubernetes.
- Access data in HDFS, Alluxio, Apache Cassandra, Apache HBase, Apache Hive, and hundreds of other data sources.

https://spark.apache.org/

Spark: Unified Big Data Analytics

- New applications of Big data workloads on Unified Engine of
  - Streaming, Batch, and Interactive.
- Composability in programming libraries for big data and encourages development of interoperable libraries
- Combining the SQL, machine learning, and streaming libraries in Spark

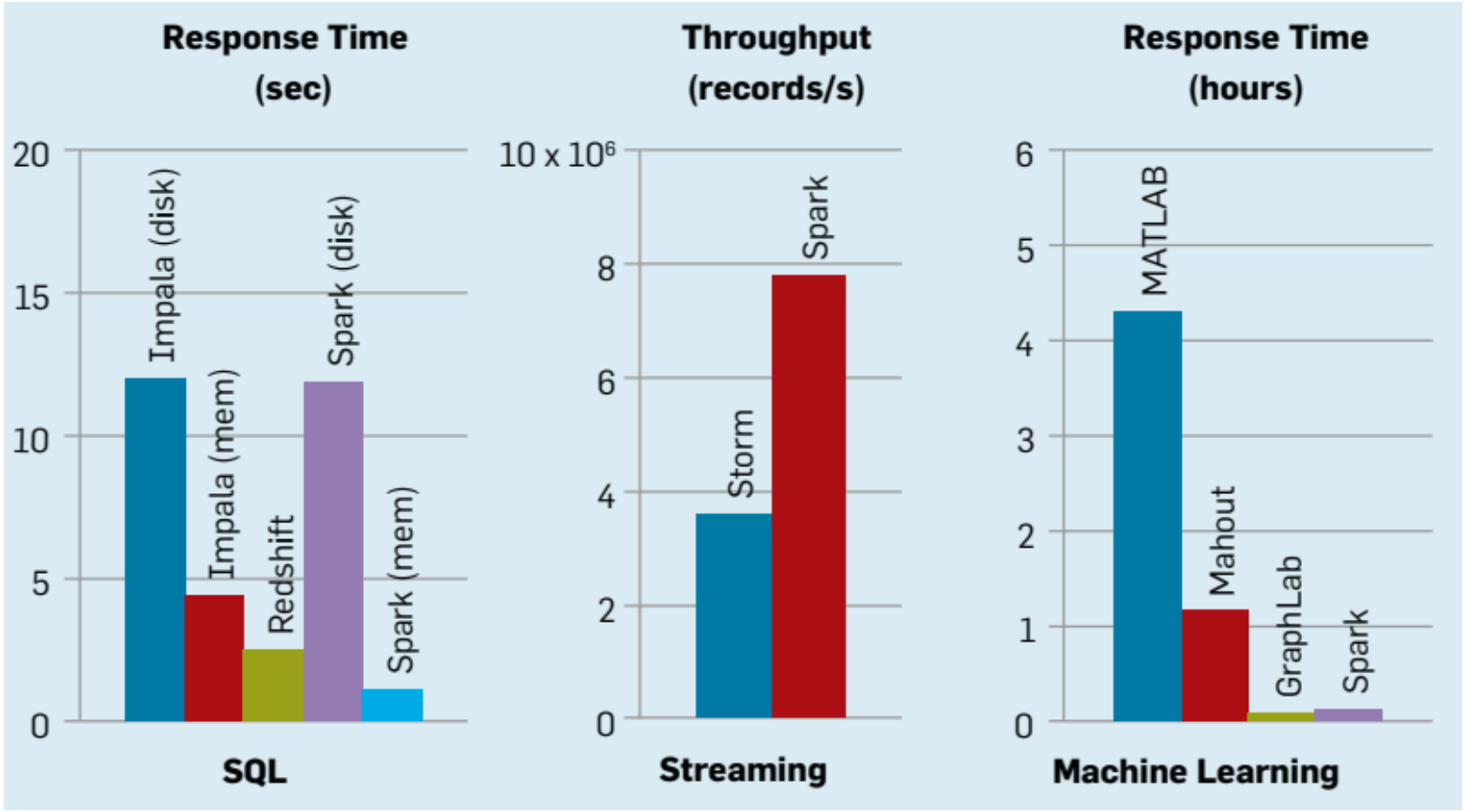
```
// Load historical data as an RDD using Spark SQL
val trainingData = sql(
   "SELECT location, language FROM old_tweets")

// Train a K-means model using MLlib
val model = new KMeans()
   .setFeaturesCol("location")
   .setPredictionCol("language")
   .fit(trainingData)
// Apply the model to new tweets in a stream
TwitterUtils.createStream(...)
      .map(tweet => model.predict(tweet.location))
```

Matei Zaharia, et al. "Apache spark: a unified engine for big data processing." *Communications of the ACM* 59.11 (2016): 56-65.

Spark: Unified Big Data Analytics

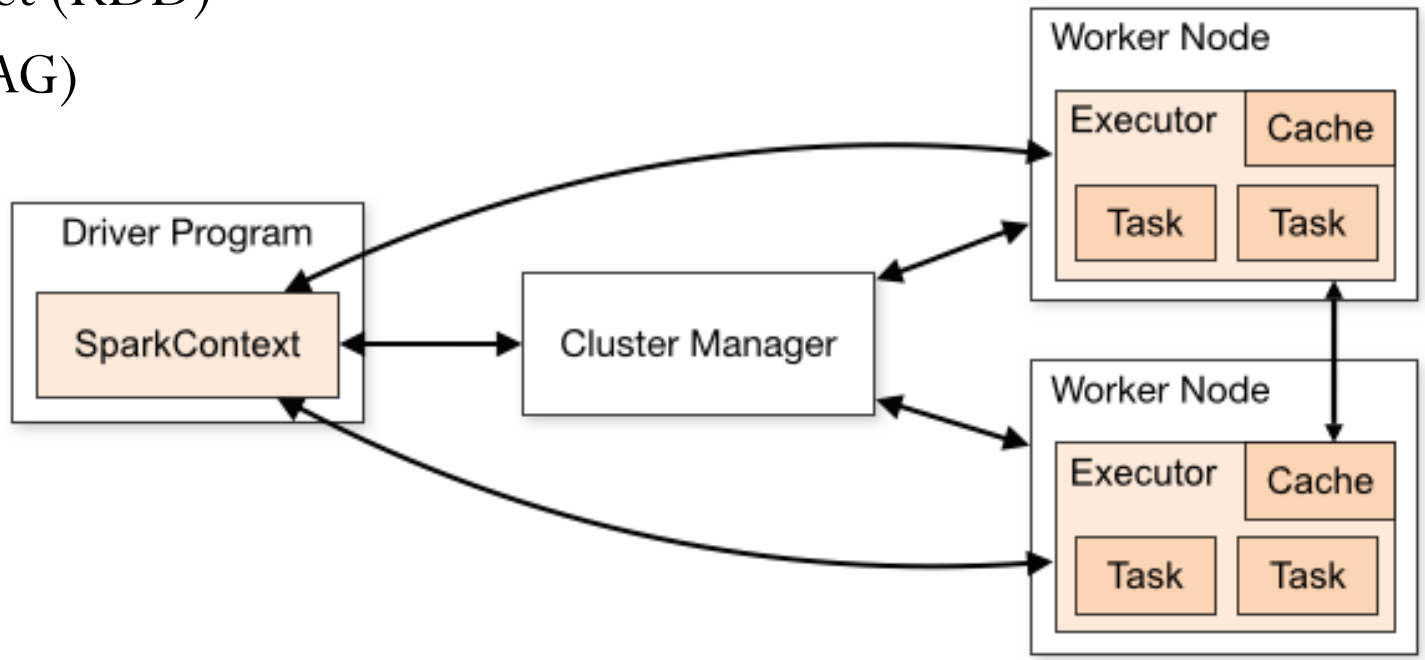
- Spark has MapReduce programming model with extended data-sharing abstraction called "Resilient Distributed Datasets," or RDDs.



Matei Zaharia, et al. "Apache spark: a unified engine for big data processing." *Communications of the ACM* 59.11 (2016): 56-65.

Spark Architecture

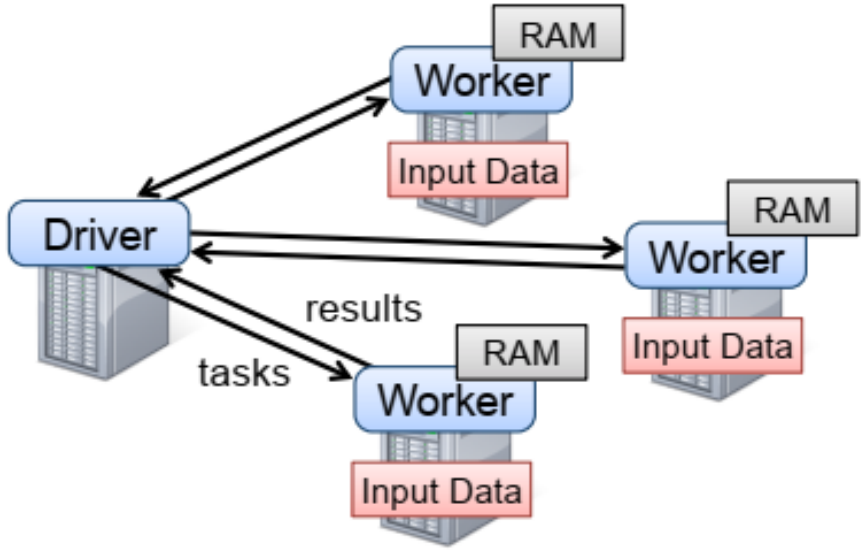
- Spark works on the popular Master-Slave architecture.
- Cluster works with a single master and multiple slaves.
- The Spark architecture depends upon two abstractions:
  - Resilient Distributed Dataset (RDD)
  - Directed Acyclic Graph (DAG)



https://spark.apache.org/docs/latest/cluster-overview.html

Resilient Distributed Datasets (RDD)

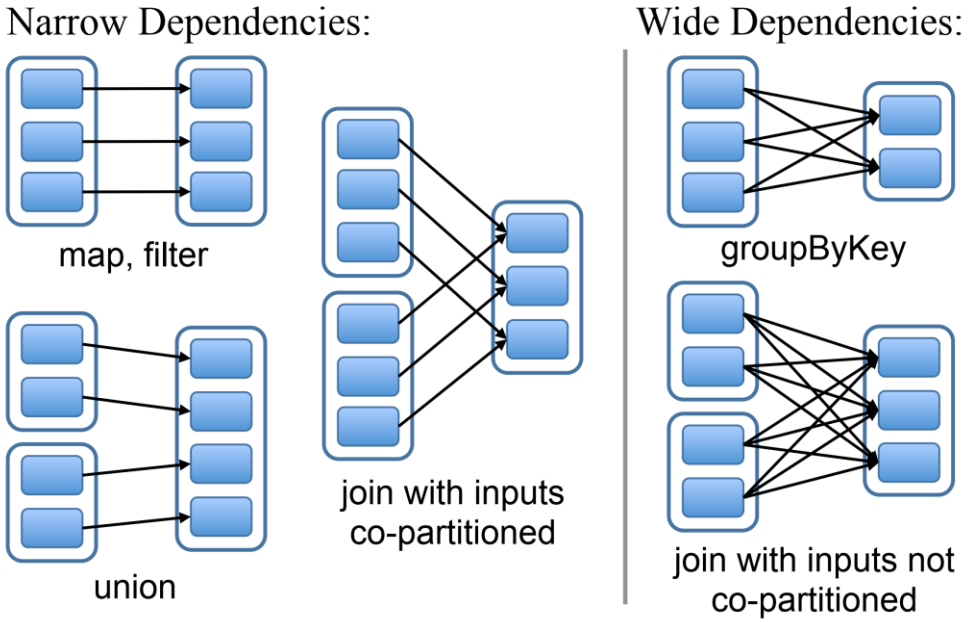
- A distributed memory abstraction: perform in-memory computations on large clusters
- Keeping data in memory can improve performance
- Spark runtime: Driver program launches multiple workers that read data blocks from a distributed file system and can persist computed RDD partitions in memory.



Matei Zaharia, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012.

Resilient Distributed Datasets (RDD)

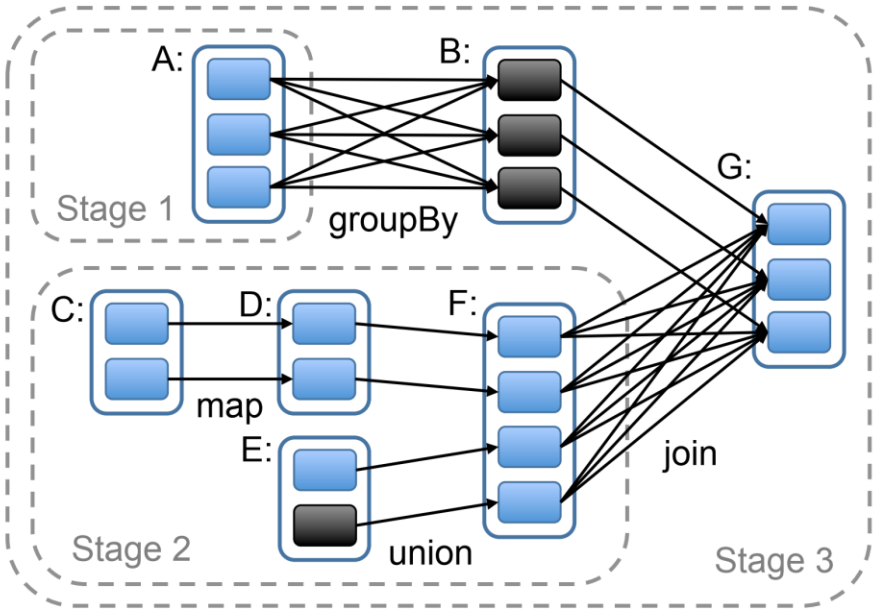
- Each box is an RDD, with partitions as shaded rectangles.
- *narrow* dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD,
- *wide* dependencies, where multiple child partitions may depend on it.



Matei Zaharia, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012.

Resilient Distributed Datasets (RDD)

- Direct Acyclic Graph (DAG) to perform a sequence of computations
- Each node is an RDD partition,
- Run an action (*e.g.*, *count* or *save*) on an RDD, the scheduler examines that RDD's lineage graph to build a DAG of *stages* to execute.
- Each stage contains with narrow dependencies
- The boundaries of the stages are the shuffle operations required for wide dependencies.
- Scheduler computes missing partitions until it computed RDD.



Matei Zaharia, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012.

Spark Code Example

- Word Count

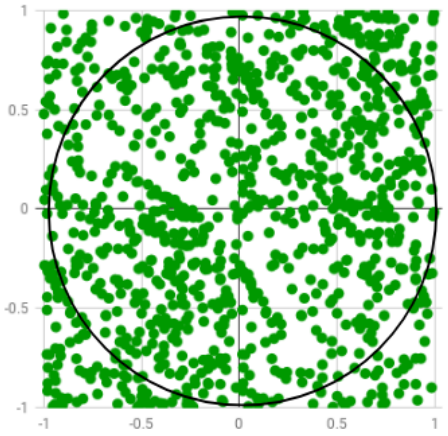
```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```

- Pi Estimation

```
List<Integer> l = new ArrayList<>(NUM_SAMPLES);
for (int i = 0; i < NUM_SAMPLES; i++) {
  l.add(i);
}

long count = sc.parallelize(l).filter(i -> {
  double x = Math.random();
  double y = Math.random();
  return x*x + y*y < 1;
}).count();
System.out.println("Pi is roughly " + 4.0 * count / NUM_SAMPLES);
```

$$\text{Pi} = 4 \times \frac{number\ of\ random\ point\ inside\ the\ circle}{number\ of\ random\ point\ inside\ the\ square}$$



Spark SQL

- Working with structured data.
- Integrated: SQL queries with Spark programs.

```
results = spark.sql("SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

- Uniform Data Access: Connect to data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC.

```
spark.read.json("s3n://…").registerTempTable("json")
results = spark.sql("""SELECT * FROM people JOIN json …""")
```

- Query and join different data sources
- Hive Integration Spark HiveQL
- Standard Connectivity: Connect through JDBC or ODBC.
- Business intelligence tools to query big data.

https://spark.apache.org/sql/

DataFrame API Examples

- Collection of data organized into named columns
- Use DataFrame API to perform various relational operations
- Automatically optimized by Spark's built-in optimizer

```
// Creates a DataFrame having a single column named "line"
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaRDD<Row> rowRDD = textFile.map(RowFactory::create);
List<StructField> fields = Arrays.asList(
  DataTypes.createStructField("line", DataTypes.StringType, true));
StructType schema = DataTypes.createStructType(fields);
DataFrame df = sqlContext.createDataFrame(rowRDD, schema);

DataFrame errors = df.filter(col("line").like("%ERROR%"));
// Counts all the errors
errors.count();
// Counts errors mentioning MySQL
errors.filter(col("line").like("%MySQL%")).count();
// Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect();
```

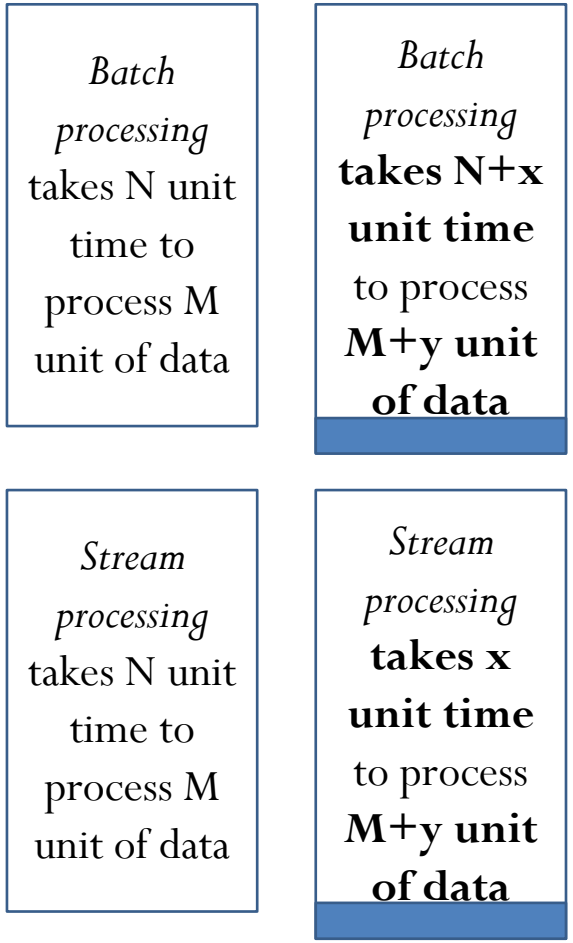
Spark Streaming

- Build scalable fault-tolerant streaming applications.
- Write streaming jobs -- Same Way -- Write batch jobs
  - Counting tweets on a sliding window

```
TwitterUtils.createStream(…)
.filter(_.getText.contains("Spark"))
.countByWindow(Seconds(5))
```

- Reuse the same code for batch processing
  - Find words with higher frequency than historic data:

```
stream.join(historicCounts).filter {
  case (word, (curCount, oldCount)) =>
    curCount > oldCount
}
```



https://spark.apache.org/streaming/

Spark GraphX

- Spark's API for graphs and graph-parallel computation

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

- Fast Speed for graph algorithms
- GraphX graph algorithms
  - PageRank
  - Connected components
  - Label propagation
  - SVD++
  - Strongly connected components
  - Triangle count

- The **joinVertices** operator joins the vertices with the input RDD and returns a new graph with the vertex properties obtained by applying the user defined map function to the result of the joined vertices.
- Vertices without a matching value in the RDD retain their original value.

https://spark.apache.org/graphx/

Spark MLlib

- Spark's scalable machine learning library
- Spark MLlib algorithms
  - Classification: logistic regression, naive Bayes,…
  - Regression: generalized linear regression, survival regression,…
  - Decision trees, Random forests, and Gradient-boosted trees
  - Recommendation: Alternating Least Squares (ALS)
  - Clustering: K-means, Gaussian mixtures (GMMs),…
  - Topic modeling: Latent Dirichlet Allocation (LDA)
  - **Frequent itemsets, Association rules, and Sequential pattern mining**

https://spark.apache.org/graphx/

Parallel Frequent Pattern Growth for Rule Mining

Apriori algorithm: Association Rule Mining

- Let I = $\{i_1, i_2, \ldots . i_m\}$ be a set of literals, called items.
- *Support* of a rule X →Y is the percentage of transactions that contain both X and Y.
- *Confidence* of a rule is percentage the if-then statements (X →Y) are found true
- Find all rules that satisfy a user-specified *minimum support* and *minimum confidence*

TID	Transaction Items
1	Bread, Jelly, PeanutButter
2	Bread, PeanutButter
3	Bread, Milk, PeanutButter
4	Beer, Bread
5	Beer, Milk

{Bread} → {PeanutButter} (Sup = 60%, Conf = 75%)

{PeanutButter} → {Bread} (Sup = 60%, Conf = 100%)

{Beer} → {Bread} (Sup = 20%, Conf = 50%)

{PeanutButter} → {Jelly} (Sup = 20%, Conf = 33.33%)

{Jelly} → {PeanutButter} (Sup = 20%, Conf = 100%)

{Jelly} → {Milk} (Sup = 0%, Conf = 0%)

Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases." *SIG-MOD*. 1993.
Ramakrishnan Srikant, and Rakesh Agrawal. "Mining Generalized Association Rules." *VLDB* 1995.

Apriori algorithm: Association Rule Mining

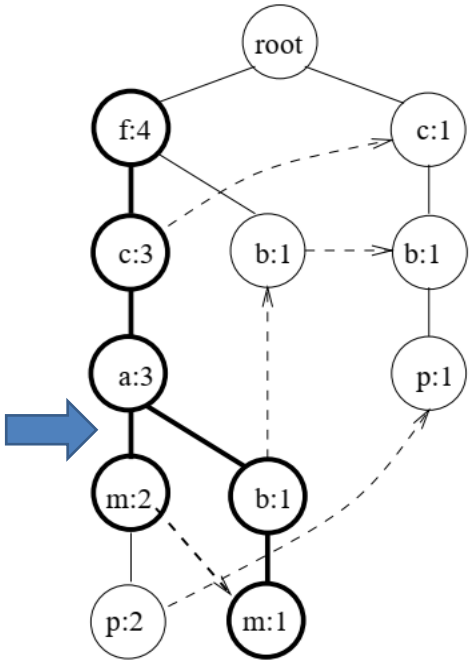
- Let I = $\{i_1, i_2, \ldots . i_m\}$ be a set of literals, called items.
- *Support* of a rule X $\rightarrow$Y is the percentage of transactions that contain both X and Y.
- *Confidence* of a rule is percentage the if-then statements (X $\rightarrow$Y) are found true
- Find all rules that satisfy a user-specified *minimum support* and *minimum confidence*
  - 75% of transactions that purchase *Bread* (antecedent) also purchase *PeanutButter* (consequent). The number 75% is the confidence factor of the rule
    - {Bread} $\rightarrow$ {PeanutButter} (Sup = 60%, Conf = 75%) (3/5 , 3/4)
    - Similarly, {PeanutButter} $\rightarrow$ {Bread} (Sup = 60%, Conf = 100%) (3/5 , 3/3)
  - 98% of customers who purchase *Tires* and *Auto accessories* also buy some *Automotive services*; here 98% is called the confidence of the rule.
    - [*Auto Accessories*], [*Tires*] $\rightarrow$ [*Automotive Services*] 98%

Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases." *SIG-MOD*. 1993.
Ramakrishnan Srikant, and Rakesh Agrawal. "Mining Generalized Association Rules." *VLDB* 1995.

FP-Growth for recommendation

- "FP" stands for Frequent Pattern in a Dataset of transactions
  1. calculate item frequencies and identify frequent items,
  2. a suffix tree (FP-tree) structure to encode transactions, and
  3. frequent itemsets can be extracted from the FP-tree.
- Input: Transaction database
- Intermediate Output: FP-Tree
- Output: {f, c, a → a, m p}, {f, c, a → b, m}

TID	Items Bought	(Ordered) Frequent Items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p



Han Jiawei, Jian Pei, and YiwenYin. "Mining frequent patterns without candidate generation." *ACM SIGMOD Record* 29.2 (2000): 1-12.

PFP: Parallel FP-Growth

- In Spark ML-Library (MLLib), a parallel version of FP-growth called PFP: Parallel FP-Growth
- PFP distributes the work of growing FP-trees based on the suffixes of transactions.
- More scalable than a single-machine implementation.
- PFP partitions computation, where each machine executes an independent group of mining tasks

Haoyuan Li, et al. "PFP: Parallel FP-Growth for query recommendation."
Proceedings of the 2008 ACM Conference on Recommender systems. 2008.

PFP: Parallel FP-Growth

Example of MapReduce FP-Growth: Five transactions composed of lower-case alphabets representing items

Map inputs (transactions) key="": value	Sorted transactions (with infrequent items eliminated)	Map outputs (conditional transactions) key: value
f a c d g i m p	f c a m p	p: f c a m m: f c a a: f c c: f
a b c f l m o	f c a b m	m: f c a b b: f c a a: f c c: f
b f h j o	f b	b: f
b c k s p	c b p	p: c b b: c
a f c e l p m n	f c a m p	p: f c a m m: f c a a: f c c: f

Reduce inputs (conditional databases) key: value		Conditional FP–trees
p:	{ f c a m / f c a m / c b }	{(c:3)} | p
m:	{ f c a / f c a / f c a b }	{ (f:3, c:3, a:3) } | m
b:	{ f c a / f / c }	{} | b
a:	{ f c / f c / f c }	{ (f:3, c:3) } | a
c:	{ f / f / f }	{ (f:3) } | c

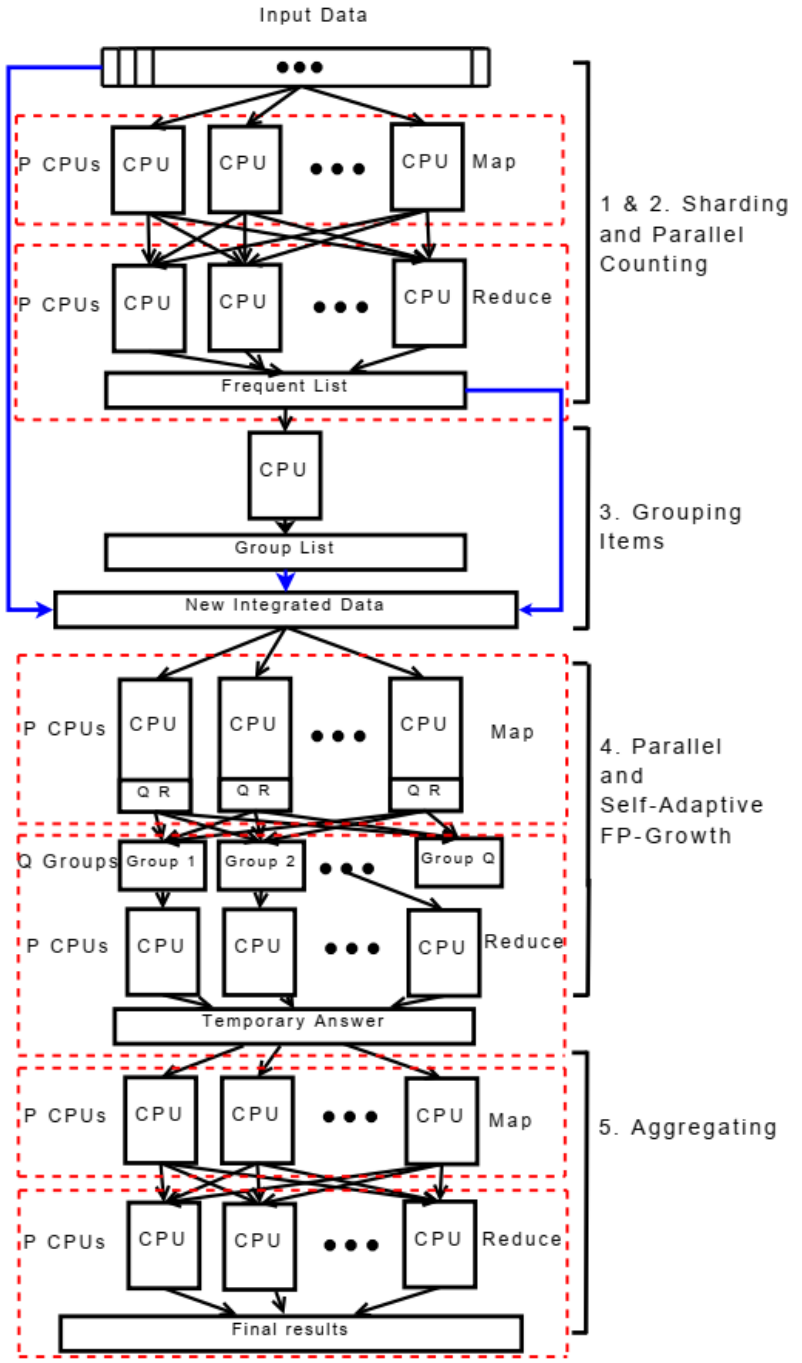
Haoyuan Li, et al. "PFP: Parallel FP-Growth for query recommendation." *Proceedings of the 2008 ACM Conference on Recommender systems*. 2008.

- **Sharding:** Divide DB into successive parts and storing the parts (as a Shard) on P different computers.
- **Parallel Counting:** MapReduce counts the support of all items that appear in DB. Each mapper inputs one shard of DB. The result is stored in F-list.
- **Grouping Items:** Dividing all the items on F-List into Q groups of a list (G-list).
- **Parallel FP-Growth:** A MapReduce

  - Mapper: Each mapper uses a Shard. It reads a transaction in the G-list and outputs one or more key-value pairs, where each key is a *group-id* and value is a **group-dependent transaction**.

  - For each *group-id*, the MapReduce groups all group-dependent transactions into a shard.

  - Reducer: Each reducer processes one or more group-dependent Shard. For each shard, a reducer builds a local FP-Tree and discover patterns.
- **Aggregating:** Aggregate the results generated as final result.



Haoyuan Li, et al. "PFP: Parallel FP-Growth for query recommendation." *Proceedings of the 2008 ACM Conference on Recommender systems*. 2008.

PFP: Parallel FP-Growth

- FP-Growth implementation takes the following (hyper-)parameters
  - minSupport: the minimum support for an itemset to be identified as frequent e.g., if an item appears 3 out of 6 transactions, it has a support of 3/6=0.5.
  - minConfidence: minimum confidence for generating Association Rule e.g., if in the transactions itemset X appears 5 times, X andY co-occur only 3 times, the confidence for the rule X =>Y is then 3/5 = 0.6.
  - numPartitions: the number of partitions used to distribute the work.
- FP-Growth model provides:
  - freqItemsets: frequent itemsets in the format of DataFrame("items"[Array], "freq"[Long])
  - associationRules: association rules generated with confidence above minConfidence, in the format of DataFrame("antecedent"[Array], "consequent"[Array], "confidence"[Double]).

https://spark.apache.org/docs/3.3.1/ml-frequent-pattern-mining.html

PFP: Parallel FP-Growth

```
List<Row> data = Arrays.asList(
  RowFactory.create(Arrays.asList("1 2 5".split(" "))),
  RowFactory.create(Arrays.asList("1 2 3 5".split(" "))),
  RowFactory.create(Arrays.asList("1 2".split(" ")))
);
StructType schema = new StructType(new StructField[]{ new StructField(
  "items", new ArrayType(DataTypes.StringType, true), false, Metadata.empty())
});
Dataset<Row> itemsDF = spark.createDataFrame(data, schema);

FPGrowthModel model = new FPGrowth()
  .setItemsCol("items")
  .setMinSupport(0.5)
  .setMinConfidence(0.6)
  .fit(itemsDF);

// Display frequent itemsets.
model.freqItemsets().show();

// Display generated association rules.
model.associationRules().show();

// transform examines the input items against all the association rules and summarize the
// consequents as prediction
model.transform(itemsDF).show();
```

https://spark.apache.org/docs/3.3.1/ml-frequent-pattern-mining.html

Ubiquitous Computing

Ubiquitous computing

- Mark Weiser: Three basic ubiquitous computing devices:
  - Tabs: a wearable device that is approx in centimeters
  - Pads: a hand-held device that is approximately a decimeter in size
  - Boards: an interactive larger display device that is approximately a meter in size
- computing is made to appear anytime and everywhere
- any device, in any location, and in any format

Weiser, Mark. "The computer for the 21st century." *ACM SIGMOBILE mobile computing and communications review* 3.3 (1999): 3-11.
https://en.wikipedia.org/wiki/Ubiquitous_computing

Edge computing

- Distributed computing paradigm
- Computation and data storage closer to the user location
- Improve response times and save bandwidth
- Cloud computing operates on big data, whereas Edge computing operates on "instant data"
- Content Delivery Network or Content Distribution Network (CDN)
- Akamai CDN: Akamai-Facebook's Photo-Serving Stack

https://en.wikipedia.org/wiki/Edge_computing

Cloudlet

- First coined by Mahadev Satyanarayanan (Satya), Victor Bahl, Ramón Cáceres, and Nigel Davie
- It is a mobility-enhanced small-scale cloud datacenter that is located at the edge of the Internet.
- It work as a *data center in a box* which *brings the cloud closer.*
- Support resource-intensive and interactive mobile applications by providing powerful computing resources to mobile devices with lower latency.

https://en.wikipedia.org/wiki/Cloudlet

Internet of Things (IoT)

- The network of physical objects —"things"— embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the Internet.
- Example:
  - "Smart Home" devices and appliances,
  - "Smart city" equipment and facilities,
- Real-Time Data Analytics
- *Information explosion* or *Data Deluge*
  - due to *data flood* or *information flood*
  - ever-increasing amount of electronic data exchanged per time unit
  - unmanageable amounts of data growth V/S power of data processing

https://en.wikipedia.org/wiki/Internet_of_things

Fog computing

- Architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing.
- Fog computing is often erroneously called edge computing, but there are key differences.
- Fog works with the cloud, whereas edge is defined by the exclusion of cloud.
- Fog is hierarchical where edge tends to be limited to a small number of layers.
- Cloud computing deal with Big Data, whereas Fog computing deals with real-time data generated by sensors or users.

IEEE Standard Association. "IEEE 1934-2018-IEEE Standard for adoption of OpenFog reference architecture for fog computing." (2018).
https://en.wikipedia.org/wiki/Fog_computing

References

- Cloud Computing: Past, Present, and Future, Professor Anthony D. Joseph, UC Berkeley Reliable Adaptive Distributed systems Lab (RAD lab) UC Berkley http://abovetheclouds.cs.berkeley.edu/
- https://en.wikipedia.org/wiki/Google_File_System
- https://sites.google.com/site/gfsassignmentwiki/home
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003.
- Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.
- Matei Zaharia, et al. "Apache spark: a unified engine for big data processing." *Communications of the ACM* 59.11 (2016): 56-65.
- Matei Zaharia, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Presented as part of *the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.
- https://spark.apache.org/docs/latest/cluster-overview.html
- https://spark.apache.org/
- https://spark.apache.org/docs/3.3.1/ml-frequent-pattern-mining.html

References

- Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases." *SIG-MOD*. 1993.
- Ramakrishnan Srikant, and Rakesh Agrawal. "Mining Generalized Association Rules." *VLDB* 1995.
- Han Jiawei, Jian Pei, and Yiwen Yin. "Mining frequent patterns without candidate generation." *ACM SIGMOD Record* 29.2 (2000): 1-12.
- Haoyuan Li, et al. "PFP: Parallel FP-Growth for query recommendation." *Proceedings of the 2008 ACM Conference on Recommender systems*. 2008.
- Weiser, Mark. "The computer for the 21st century." *ACM SIGMOBILE Mobile Computing and Communications Review* 3.3 (1999): 3-11.
- https://en.wikipedia.org/wiki/Ubiquitous_computing
- https://en.wikipedia.org/wiki/Edge_computing
- https://en.wikipedia.org/wiki/Cloudlet
- IEEE Standard Association. "IEEE 1934-2018-IEEE Standard for adoption of OpenFog reference architecture for fog computing." (2018).
- https://en.wikipedia.org/wiki/Fog_computing

ขอบคุณ
Thai

Grazie
Italian

תודה רבה
Hebrew

धन्यवादः
Sanskrit

ಧನ್ಯವಾದಗಳು
Kannada

Ευχαριστώ
Greek

Thank You
English

Gracias
Spanish

Спасибо
Russian

Obrigado
Portuguese

شكراً
Arabic

https://sites.google.com/site/animeshchaturvedi07

Merci
French

多謝
Traditional
Chinese

धन्यवाद
Hindi

Danke
German

多谢
Simplified
Chinese

நன்றி
Tamil
Tamil

ありがとうございました
Japanese

감사합니다
Korean