CAIM: Cerca i Anàlisi d'Informació Massiva

Grau en Enginyeria Informàtica, FIB

Tema 6: Big Data, Mapreduce, NoSQL

Slides by David García-Soriano, Marta Arias, José Luis Balcázar, Ramon Ferrer-i-Cancho, Ricard Gavaldà
Department of Computer Science, UPC

Fall 2025

Architecture of large-scale systems: Outline

1. Cluster architectures.
2. Distributed file systems.
3. Mapreduce and Hadoop.

The Big Data revolution

Modern systems collect vast amounts of data continously.

- Internet traffic: a backbone router can move >3 Tb/s
- Astronomy: the Vera Rubin Observatory captures 20 TB per night (1 TB=10^{12} bytes)
- CERN's LHC detectors generate up to 1 PB/s $(10^{15} b/s)$ of raw signals (most of which is discarded immediately!)
- Autonomous driving: A Tesla vehicle generates several terabytes of sensor data *daily*.
- How to **store** and **analyze** so much data?

Big Data

- Datasets larger what typical storage tools can handle
- The 3 V's: Volume (amount), Velocity (processing speed), Variety (number of types).
- Figure that grows concurrently with technology
- The problem has always existed, and has always driven innovation
- Technological problem:
  - how to store, use & analyze?
- Or business problem?
  - what to look for in the data?
  - what questions to ask?
  - how to model the data?
  - where to start?

Example: web search engines

Google in 1998:

- 24 million web pages; 148 GB of text
- 14 million terms in lexicon; fits in 256 MB RAM
- 37 GB inverted index
- 3 crawlers; 100 webpages crawled / second
- Anticipated hitting O.S. limits at about 100 million pages

Today (exact figures undisclosed):

- Massive vector embeddings for modern AI-powered search
- 100s petabytes transferred per day? (1PB = 10^{15} bytes)
- 100s exabytes of storage? (1 EB = 10^{18} bytes)
- Millions of servers

S.Brin, L.Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine.

Computer Networks and ISDN Systems 30, 1998.

Supercomputer vs. Commodity Cluster

Supercomputer

- **Goal:** Solve a single, complex problem (e.g., climate simulation).
- **Hardware:** Custom, massive CPU & RAM.
- **Network:** High-speed, low-latency.
- **Faults:** Hardware is assumed to be reliable.

Supercomputer vs. Commodity Cluster

Supercomputer

- **Goal:** Solve a single, complex problem (e.g., climate simulation).
- **Hardware:** Custom, massive CPU & RAM.
- **Network:** High-speed, low-latency.
- **Faults:** Hardware is assumed to be reliable.

Cluster

- **Goal:** Solve a large problem by splitting it into independent subtasks.
- **Hardware:** Thousands of cheap, standard PCs.
- **Network:** Slow, standard, often a bottleneck.
- **Faults:** Hardware is assumed to *fail constantly*. Fault tolerance is managed by *software*.

Google’s approach to Big Data

- Many machines, many data centers, many programmers
- Huge & complex data
- Need for abstraction layers

Google's approach to Big Data

- Many machines, many data centers, many programmers
- Huge & complex data
- Need for abstraction layers

Published several influential proposals:

- Hardware abstraction: The Google Cluster architecture (IEEE Micro, 2003)
- Data abstraction:

  The Google File System (SOSP, 2003)

  BigTable (OSDI, 2006)
- Programming model for parallelism: MapReduce (OSDI, 2004)

Cluster architecture: design criteria

Use many cheap machines, rather than expensive servers

- High task parallelism; Little instruction parallelism

  (e.g., process posting lists, summarize docs)
- Peak processor performance less important than price/performance

  price is superlinear in performance!

Cluster architecture: design criteria

Use many cheap machines, rather than expensive servers

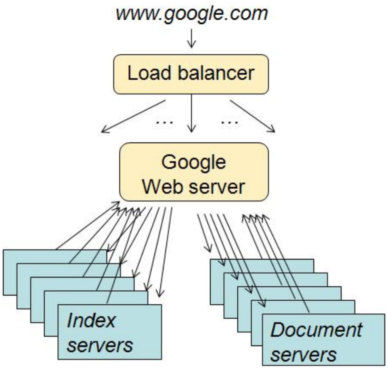
- High task parallelism; Little instruction parallelism

  (e.g., process posting lists, summarize docs)
- Peak processor performance less important than price/performance

  price is superlinear in performance!
- Commodity-class PCs. Cheap, easy to replace
- Optimize for request throughput, not response time
- Redundancy
- Reliability comes from replicating services across machines; managed by software
- Servers are short-lived anyway (< 3 years)

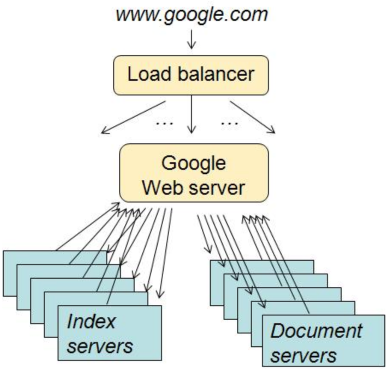
L.A. Barroso, U. Hölzle, P. Ranganathan: The Datacenter as a Computer. Springer, 2019

Example: web search cluster



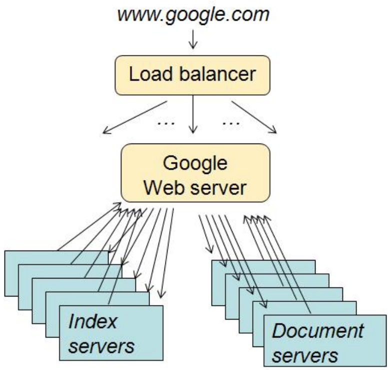
- Load balancer chooses freest / closest GWS

Example: web search cluster



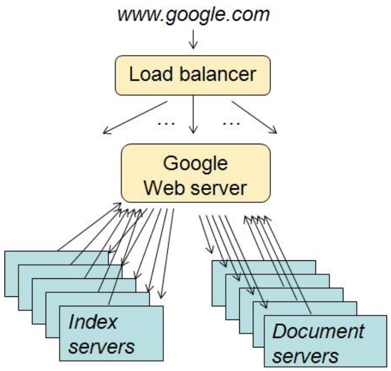
- Load balancer chooses freest / closest GWS
- GWS asks several index servers

Example: web search cluster



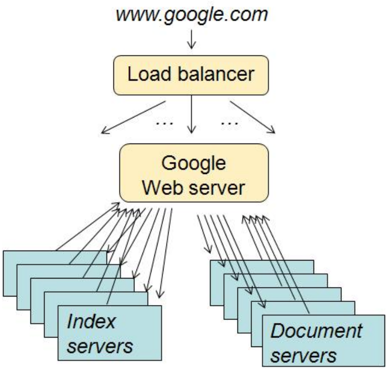
- Load balancer chooses freest / closest GWS
- GWS asks several index servers
- They compute hit lists for query terms, intersect them, and rank them

Example: web search cluster



- Load balancer chooses freest / closest GWS
- GWS asks several index servers
- They compute hit lists for query terms, intersect them, and rank them
- Answer (docid list) is returned to GWS

Example: web search cluster



- Load balancer chooses freest / closest GWS
- GWS asks several index servers
- They compute hit lists for query terms, intersect them, and rank them
- Answer (docid list) is returned to GWS
- GWS asks document servers for a query-specific summary, url, etc.
- GWS formats an html page & returns to user

L.A. Barroso, J. Dean, U. Hölzle: Web Search for a Planet: The Google Cluster Architecture, IEEE Micro, 2003

Web search: Index “shards”

- The inverted index can be very large.
- Documents randomly distributed into *shards*
- The index is divided into *index shards*, each built for a subset of the documents
- Several replicas (index servers) for each index shard

Web search: Index “shards”

- The inverted index can be very large.
- Documents randomly distributed into *shards*
- The index is divided into *index shards*, each built for a subset of the documents
- Several replicas (index servers) for each index shard
- Queries routed through local load balancer
- For speed & fault tolerance
- Most accesses are read-only; updates are infrequent, unlike traditional DB’s

Distributed File Systems (DFS): design goals

- **Problem:** Store huge files on thousands of cheap, unreliable PCs.
- **Implementations:**

Proprietary:	Google File System (GFS, 2003); succeeded by Colossus in 2010.
Open-source:	Hadoop Distributed File System (HDFS, 2006)

Distributed File Systems (DFS): design goals

- **Problem:** Store huge files on thousands of cheap, unreliable PCs.
- **Implementations:**

Proprietary:	Google File System (GFS, 2003); succeeded by Colossus in 2010.
Open-source:	Hadoop Distributed File System (HDFS, 2006)

- **Assumptions:**
  - Hardware failure is the norm.
  - Optimized for large files (TBs).
  - Workload is **write-once, read-many**.
  - Prefer **high sustained throughput** to low latency.

S. Ghemawat, H. Gobioff, Sh.-T. Leung: The Google File System, SOSP 2003
Apache: HDFS Architecture Guide

D. Hildebrand, D. Serenyi: Colossus under the hood: a peek into Google's scalable storage system, 2021.

Distributed File Systems (DFS): architecture

- **Master Node** (GFS Master / HDFS NameNode)
  - Manages metadata (directory tree, file-to-chunk mapping, permissions. . .) by polling the workers.
  - Tells the client which worker to get the data from.
  - Monitors worker health via "heartbeat"messages.

Distributed File Systems (DFS): architecture

- **Master Node** (GFS Master / HDFS NameNode)
  - Manages metadata (directory tree, file-to-chunk mapping, permissions. . .) by polling the workers.
  - Tells the client which worker to get the data from.
  - Monitors worker health via "heartbeat"messages.

- **Worker Nodes** (GFS ChunkServer / HDFS DataNode)
  - Store the actual data on local disks.

Distributed File Systems (DFS): architecture

- **Master Node** (GFS Master / HDFS NameNode)
    - Manages metadata (directory tree, file-to-chunk mapping, permissions. . .) by polling the workers.
    - Tells the client which worker to get the data from.
    - Monitors worker health via "heartbeat"messages.
- **Worker Nodes** (GFS ChunkServer / HDFS DataNode)
    - Store the actual data on local disks.
- **Fault Tolerance Model:**
    - Files are split into large **chunks** (typically 64 or 128 MB) with unique ids.
    - Each chunk is **replicated** (3x) on different worker nodes.
    - Replicas are **rack-aware**: the system tries to place replicas on different machine racks.

MapReduce and Hadoop

- Mapreduce: Large-scale programming model developed at Google (2004)
  - Proprietary implementation
  - Implements old ideas from functional programming, distributed systems, DB's . . .

MapReduce and Hadoop

- Mapreduce: Large-scale programming model developed at Google (2004)
  - Proprietary implementation
  - Implements old ideas from functional programming, distributed systems, DB's ...
- Hadoop: Open source implementation at Yahoo! (2006)
  - HDFS: Open Source analog of GFS
    - newer systems use storage services (Amazon S3, Google Cloud GCS, Azure ADLS)
    - compute is decoupled from storage; HDFS remains common on-prem.
  - Hive (Facebook): SQL on top of Hadoop
  - Spark: Open Source engine for large-scale data analytics $\rightarrow$ industry standard

J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004.

MapReduce and Hadoop

Design goals:

- Scalability to large data volumes and number of machines
  - 1000's of machines, 10,000's disks
  - Abstract hardware & distribution
  - Easy to use: good learning curve for programmers
- Cost-efficiency:
  - Commodity machines: cheap, but unreliable
  - Commodity network
  - Automatic fault-tolerance and tuning. Fewer administrators

The MapReduce Programming Model

- Data type: (key, value) records

The MapReduce Programming Model

- Data type: (key, value) records
- `Map` function:

$$(K_{ini}, V_{ini}) \rightarrow \text{list}\langle (K_{inter}, V_{inter}) \rangle$$

The MapReduce Programming Model

- Data type: (key, value) records
- `Map` function:

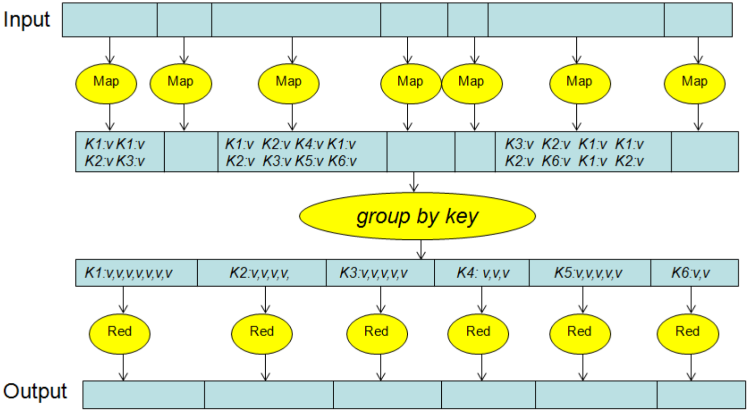
$$(K_{ini}, V_{ini}) \rightarrow \text{list}\langle (K_{inter}, V_{inter}) \rangle$$

- `Reduce` function:

$$(K_{inter}, \text{list}\langle V_{inter} \rangle) \rightarrow \text{list}\langle (K_{out}, V_{out}) \rangle$$

Semantics

Key step, handled by the platform: group by or shuffle by key



Example 1: Word Count

Input: A big file with many lines of text
Output: For each word, # times it appears in the file

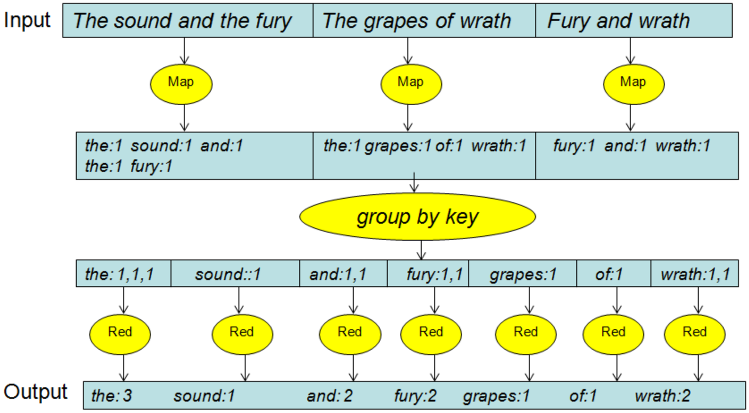
Example 1: Word Count

Input: A big file with many lines of text
Output: For each word, # times it appears in the file

```
map(line):
    foreach word in line.split() do
        output (word,1)

reduce(word,L):
    output (word,sum(L))
```

Example 1: Word Count



Example 2: Temperature statistics

Input: Set of files with records `(time,place,temperature)`
Output: For each place, report maximum, minimum, and average temperature

Example 2: Temperature statistics

Input: Set of files with records `(time,place,temperature)`
Output: For each place, report maximum, minimum, and average temperature

```
map(file):
    foreach record (time,place,temp) in file do
        output (place,temp)

reduce(p,L):
    output (p,(max(L),min(L),sum(L)/length(L)))
```

Example 3: Numerical integration

Input: A function $f : R \to R$, an interval $[a, b]$
Output: An approximation of the integral of f in $[a, b]$

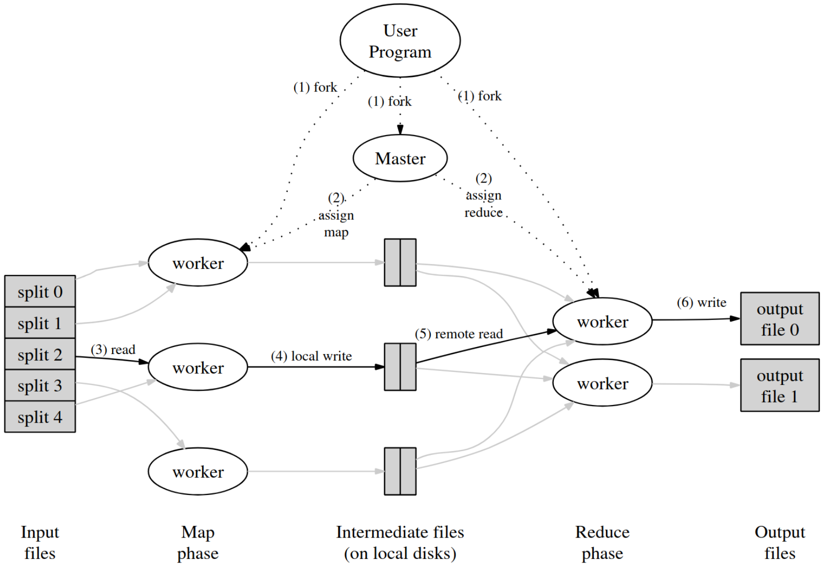
Example 3: Numerical integration

Input: A function $f : R \to R$, an interval $[a, b]$
Output: An approximation of the integral of f in $[a, b]$

```
map(start,end):
    sum = 0;
    for (x = start; x < end; x += step)
        sum += f(x)*step;
    output (0,sum)

reduce(key,L):
    output (0,sum(L))
```

Mapreduce Implementation (I)



Mapreduce Implementation (II)

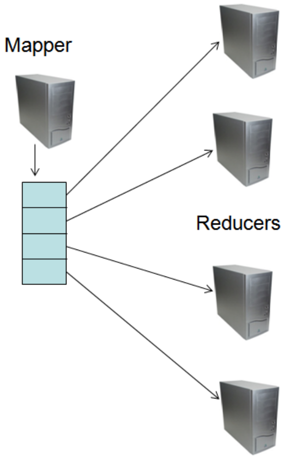
- Some *mapper* machines, some *reducer* machines
- Instances of *map* distributed to mappers
- Instances of *reduce* distributed to reduce
- Platform takes care of shuffling through network
- Dynamic load balancing

Mapreduce Implementation (II)

- Some *mapper* machines, some *reducer* machines
- Instances of *map* distributed to mappers
- Instances of *reduce* distributed to reduce
- Platform takes care of shuffling through network
- Dynamic load balancing
- Mappers write their output to local disk (not HDFS)
- If a map or reduce instance fails, automatically reexecuted
- Incidentally, information may be sent compressed

Mapreduce Implementation (III)

- A mapper writes to local disk
- In fact, makes as many partitions as reducers
- Keys are distributed to reducers by hashing the key
- Alternatively, the programmer may specify a `Partition` function



An Optimization: Combiner

- `map` outputs pairs `(key,value)`
- `reduce` receives pair `(key,list-of-values)`
- `combiner(key,list-of-values)` is applied to mapper output, *before* shuffling
- may help sending much less information
- must be associative and commutative

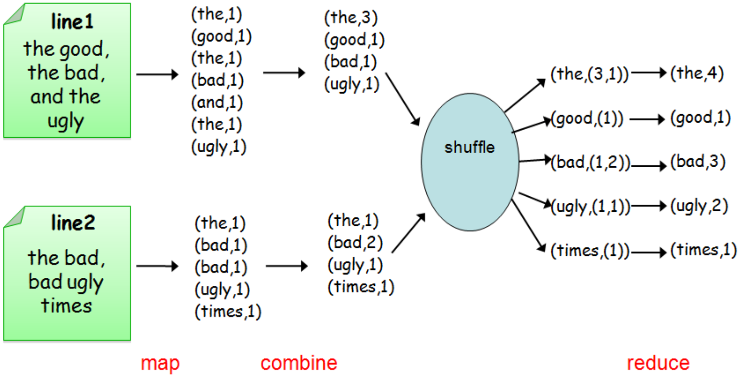
Example 1: Word Count, revisited

```
map(line):
    foreach word in line.split() do
        output (word,1)

combine(word,L):
    output (word,sum(L))

reduce(word,L):
    output (word,sum(L))
```

Example 1: Word Count,revisited



Example 4: Inverted Index

Input: A set of text files
Output: For each word, the list of files that contain it

Example 4: Inverted Index

Input: A set of text files
Output: For each word, the list of files that contain it

```
map(filename):
    foreach word in the file text do
        output (word, filename)
```

Example 4: Inverted Index

Input: A set of text files
Output: For each word, the list of files that contain it

```
map(filename):
    foreach word in the file text do
        output (word, filename)

combine(word,L):
    remove duplicates in L;
    output (word,L)
```

Example 4: Inverted Index

Input: A set of text files
Output: For each word, the list of files that contain it

```
map(filename):
    foreach word in the file text do
        output (word, filename)

combine(word,L):
    remove duplicates in L;
    output (word,L)

reduce(word,L):
    //want sorted posting lists
    output (word,sort(L))
```

Can also keep pairs (filename,frequency)

Example 5. Sorting

Input: A set S of elements of a type T with a $<$ relation
Output: The set S, sorted

1. `map(x): output x`
2. `Partition`: any such that `k` $<$ `k'` $\rightarrow$ `Partition(k)` $\leq$ `Partition(k')`
3. Now each reducer gets an interval of T according to $<$ (e.g., 'A'..'F', 'G'..'M', 'N'..'S','T'..'Z')
4. Each reducer sorts its list

Example 6: Entropy of a distribution

Input: A multiset S
Output: The entropy of S:

$$H(S) = \sum_i -p_i \log(p_i),$$

where $p_i = \frac{f_i}{|S|}$ and f_i is the frequency of the ith item.

Example 6: Entropy of a distribution

Input: A multiset S
Output: The entropy of S:

$$H(S) = \sum_i -p_i \log(p_i),$$

where $p_i = \frac{f_i}{|S|}$ and f_i is the frequency of the ith item.

Need to split into several jobs:

1. Compute absolute counts per item and total count.
2. Compute frequencies, summands, and total entropy.

Example 6: Entropy of a distribution

Job 1: Compute absolute counts and total count

- `map(i):`

```
    output (i, 1)
    output ("TOTAL", 1)    # a special key
```

- `reduce(key, L):`

Example 6: Entropy of a distribution

Job 1: Compute absolute counts and total count

- `map(i):`

```
    output (i, 1)
    output ("TOTAL", 1)    # a special key
```

- `reduce(key, L):`

```
    output (key, sum(L))
```

Example 6: Entropy of a distribution

Job 1: Compute absolute counts and total count

- `map(i):`

```
    output (i, 1)
    output ("TOTAL", 1)    # a special key
```

- `reduce(key, L):`

```
    output (key, sum(L))
```

Job 2: Compute entropy sum

- First, read the total $N = \sum_i f_i$ from Job 1's output.
- `map(key, value):`

```
    if key != "TOTAL":
      p_key = value / N
      output (“summand”, -p_key * log(p_key))
```

Example 6: Entropy of a distribution

Job 1: Compute absolute counts and total count

- `map(i):`

```
    output (i, 1)
    output ("TOTAL", 1)    # a special key
```

- `reduce(key, L):`

```
    output (key, sum(L))
```

Job 2: Compute entropy sum

- First, read the total $N = \sum_i f_i$ from Job 1's output.
- `map(key, value):`

```
    if key != "TOTAL":
      p_key = value / N
      output ("summand", -p_key * log(p_key))
```

- `reduce("summand" L):`

```
    output ("entropy" sum(L))
```

Example 7: Natural Join of Database Tables

- **Input:** Two relations, $R(A, B)$ and $S(B, C)$.
- **Output:** $R \bowtie S = \{(a, b, c) \mid (a, b) \in R \wedge (b, c) \in S\}$.

Example 7: Natural Join of Database Tables

- **Input:** Two relations, $R(A, B)$ and $S(B, C)$.
- **Output:** $R \bowtie S = \{(a, b, c) \mid (a, b) \in R \wedge (b, c) \in S\}$.
- map(tuple, origin):

```
  if tuple is $(a, b)$ from R:
    output (b, ("R", a))
  if tuple is $(b, c)$ from S:
    output (b, ("S", c))
```

Example 7: Natural Join of Database Tables

- **Input:** Two relations, $R(A, B)$ and $S(B, C)$.
- **Output:** $R \bowtie S = \{(a, b, c) \mid (a, b) \in R \wedge (b, c) \in S\}$.
- map(tuple, origin):

```
  if tuple is $(a, b)$ from R:
    output (b, (“R”, a))
  if tuple is $(b, c)$ from S:
    output (b, (“S”, c))
```

- reduce(b, L):

```
  R_list = [ a | (“R”, a) $\in$ L ]
  S_list = [ c | (“S”, c) $\in$ L ]
  for a in R_list:
    for c in S_list:
      output (a, b, c)
```

Mapreduce/Hadoop: Conclusion

- One of the basis for the Big Data / NoSQL revolution
- Was the open-source standard big data distributed processing for one decade
- Abstracts from cluster details
- Missing features can be externally added
  - Scripting languages, workflow management, SQL-like languages. . .

Cons:

- Complex to setup, lengthy to program
- Input and output of each job goes to disk; slow
- No support for online, streaming processing; superseeded

Beyond Hadoop: Online, real-time

Apache Kafka: Massive scale message distributing systems

Apache Storm: Distributed stream processing computation framework; designed for low latency

Apache Spark: In-memory, interactive, real-time. Can handle batch, streaming, SQL, ML...

Hadoop vs. Spark: Disk vs. Memory

[source: https://www.tutorialspoint.com/apache_spark/apache_spark_pdf_version.htm]

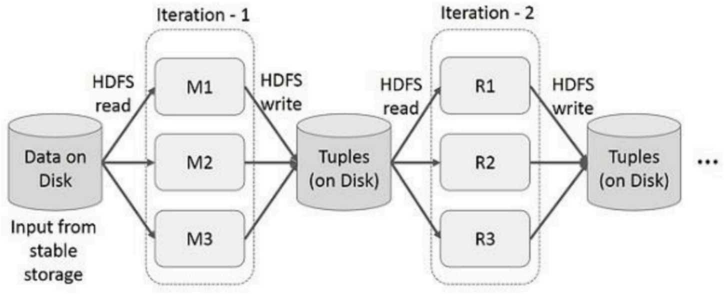


Figure: Iterative operations on MapReduce

Hadoop vs. Spark: Disk vs. Memory

[source: https://www.tutorialspoint.com/apache_spark/apache_spark_pdf_version.htm]

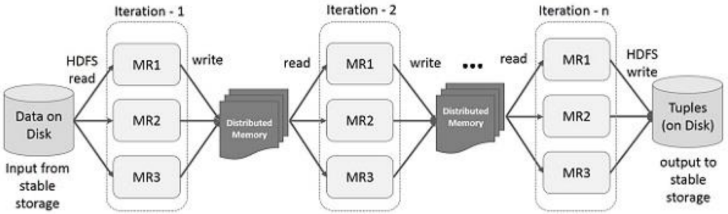


Figure: Iterative operations on Spark RDD

Two Key Concepts in Spark

- Resilient Distributed Datasets (RDD)
  - Dataset partitioned among worker nodes (in RAM)
  - Can be created from HDFS files
- Directed Acyclic Graph (DAG)
  - Specifies data transformations
  - Data moves from one state to another
- Avoid one of Hadoop’s bottlenecks: disk writes
- Allow for efficient stream processing

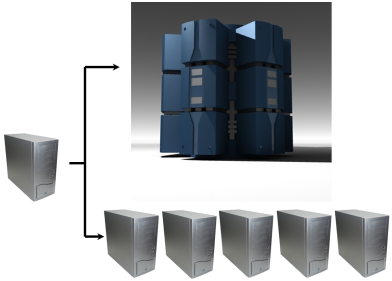
NoSQL: Outline

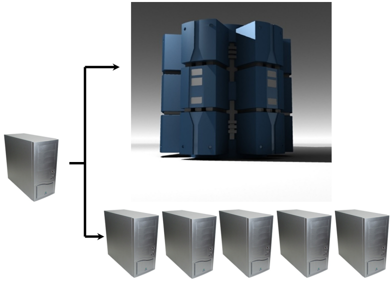
1. NoSQL: Generalities
2. The CAP Theorem
3. Key-value DB's: Dynamo and Cassandra
4. A document-oriented DB: MongoDB

The problem with Relational DBs

- The relational DB has ruled for 2-3 decades
- Superb capabilities, superb implementations
- One of the ingredients of the web revolution
  - LAMP = Linux + Apache HTTP server + MySQL + PHP
- Main problem: scalability

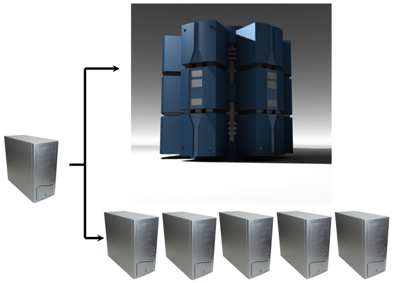
Scaling UP (more power)





Scaling UP (more power)

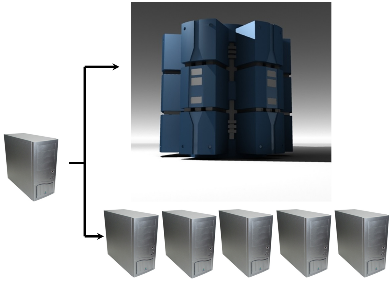
- Price superlinear in performance & power
- Performance ceiling



Scaling UP (more power)

- Price superlinear in performance & power
- Performance ceiling

Scaling OUT (more machines)



Scaling UP (more power)

- Price superlinear in performance & power
- Performance ceiling

Scaling OUT (more machines)

- No performance ceiling, but
- More complex management
- More complex programming
- Problems keeping ACID properties (atomicity, consistency, isolation, and durability)

The problem with Relational DBs

- RDBMS scale *up* well (single node). Don't scale *out* well
- Vertical partitioning: Different tables in different servers
- Horizontal partitioning: Rows of same table in different servers

The problem with Relational DBs

- RDBMS scale *up* well (single node). Don't scale *out* well
- Vertical partitioning: Different tables in different servers
- Horizontal partitioning: Rows of same table in different servers

Apparent solution: Replication and caches

- Good for fault-tolerance, for sure
- OK for many concurrent reads

The problem with Relational DBs

- RDBMS scale *up* well (single node). Don't scale *out* well
- Vertical partitioning: Different tables in different servers
- Horizontal partitioning: Rows of same table in different servers

Apparent solution: Replication and caches

- Good for fault-tolerance, for sure
- OK for many concurrent reads
- Not much help with writes, if we want to keep ACID

There’s a reason: The CAP theorem

Three desirable properties:

- Consistency: After an update to the object, every access to the object will return the updated value
- Availability: At all times, all DB clients are able to access *some* version of the data. Equivalently, every request receives an answer
- Partition tolerance: The DB is split over multiple servers communicating over a network. Messages among nodes may be lost arbitrarily

There’s a reason: The CAP theorem

Three desirable properties:

- Consistency: After an update to the object, every access to the object will return the updated value
- Availability: At all times, all DB clients are able to access *some* version of the data. Equivalently, every request receives an answer
- Partition tolerance: The DB is split over multiple servers communicating over a network. Messages among nodes may be lost arbitrarily

The CAP theorem [Brewer 00, Gilbert-Lynch 02] says:

Not distributed system can have these three properties

In other words: In a system made up of nonreliable nodes and network, it is impossible to implement atomic reads & writes and ensure that every request has an answer.

CAP theorem: Proof

- Two nodes, A, B
- A gets request “read(x)”
- To be consistent, A must check whether some “write(x,value)” performed on B
- . . . so sends a message to B
- If A doesn’t hear from B, either A answers (inconsistently)
- or else A does not answer (not available)

The problem with RDBMS

- A truly distributed, truly relational DBMS should have Consistency, Availability, and Partition Tolerance
- . . . which is impossible

The problem with RDBMS

- A truly distributed, truly relational DBMS should have Consistency, Availability, and Partition Tolerance
- . . . which is impossible
- Relational is not distributed! (full C+A at the cost of P)
  - CA: Single-node RDBMS (MySQL, PostgreSQL)
- NoSQL obtains scalability by going for A+P or for C+P
  - CP: HBase / BigTable, MongoDB, Redis
  - AP: Cassandra, DynamoDB
- . . . and as much of the third one as possible

BASE, eventual consistency

- Basically Available, Soft state, Eventual consistency
- Eventual consistency: If no new updates are made to an object, eventually all accesses will return the last updated value.
- ACID is pessimistic. BASE is optimistic. Accepts that DB consistency will be in a state of flux
- Surprisingly, OK with many applications
- And allows *far* more scalability than ACID

NoSQL: properties

Properties of most NoSQL DB's:

1. BASE instead of ACID
2. Simple queries. No joins
3. No schema
4. Decentralized, partitioned (even multi data center)
5. Linearly scalable using commodity hardware
6. Fault tolerance
7. Not for online (complex) transaction processing

A key-value store: Dynamo

- Amazon's propietary system
- Key-value store: "just" a distributed hash table.
  - can't do range searches.
- Very influential: Riak, Cassandra, Voldemort
- Goal: system where ALL customers have a good experience, not just the majority
- I.e., very high **availability**

Dynamo

Interesting feature:

- In most rdbms, conflicts resolved at write time, so read remains simple.
- That's why lock before write. “Syntactic” resolution
- In Dynamo, conflict resolution at reads – “semantic” – solved by client with business logic

Dynamo

Interesting feature:

- In most rdbms, conflicts resolved at write time, so read remains simple.
- That's why lock before write. "Syntactic" resolution
- In Dynamo, conflict resolution at reads – "semantic" – solved by client with business logic

Example:

- Client gets several versions of end-user's shopping cart
- Knowing their business, decides to merge; no item ever added to cart is lost, but deleted items may reappear
- Final purchase we want to do in full consistency

Column-Family Stores: Bigtable & HBase

- Bigtable (Google): The paper that defined the wide-column model.
  - A sparse, distributed, **persistent** multi-dimensional map.
  - Persistent $\rightarrow$ allows “time-travel” to the past.
  - Data is sorted and row key, but column families are stored in separat files.
- HBase (Apache): The open-source implementation inspired by Bigtable.
  - Runs on top of HDFS (for data storage) and ZooKeeper (for coordination).
  - A classic CP (Consistent, Partition-Tolerant) system.
  - Uses probabilistic data structures (Bloom Filters) to quickly determine if a row-column pair exists.

Cassandra

- Key-value pairs, like Dynamo, Riak, Voldemort
- But also richer data model: Columns and Supercolumns
- Write-optimized

  Choice if you write more than you read, such as logging

A document-oriented DB: MongoDB

- Richer data model than most NoSQL DB's
- More flexible queries than most NoSQL DB's
- No schemas, allowing for dynamically changing data
- Indexing
- MapReduce & other aggregations
- Stored JavaScript functions on server side
- Automatic sharding and load balancing
- Javascript shell

MongoDB Data model

- Document: Set of key-value pairs and embedded documents
- Collection: Group of documents
- Database: A set of collections + permissions + . . .

Relational analogy:
Collection = table; Document = row

Example Document

```
{
    "name" : "Anna Rose",
    "profession" : "lawyer",
    "address" : {
        "street" : "Champs Elisees 652",
        "city" : "Paris",
        "country" : "France"
    }
}
```

Always an extra field `_id` with unique value

Consistency in MongoDB

- By default, all operations are “fire-and-forget”: client does not wait until finished
- Allows for very fast reads and writes
- Price: possible inconsistencies

Consistency in MongoDB

- By default, all operations are “fire-and-forget”: client does not wait until finished
- Allows for very fast reads and writes
- Price: possible inconsistencies

- Operations can be made *safe*: wait until completed
- Price: client slowdown

Sharding in MongoDB

- With a shard key, a user tells how to split DB into shards
- E.g. `"name"` as a shard key may split `db.people` into 3 shards A-G, H-R, S-Z, sent to 3 machines
- Random shard keys good idea

Sharding in MongoDB

- With a shard key, a user tells how to split DB into shards
- E.g. `"name"` as a shard key may split `db.people` into 3 shards A-G, H-R, S-Z, sent to 3 machines
- Random shard keys good idea

- Shards themselves may vary over time to balance load
- E.g., if many A's arrive the above may turn into A-D, E-P, Q-Z

NoSQL: Conclusions

- Relational DBs (ACID) scale up.
- Big Data needs (Volume, Velocity) require scaling out.
- The **CAP Theorem** forces us to choose: any distributed system (P) must sacrifice Consistency or Availability.
- **NoSQL** NoSQL databases offer different trade-offs to achieve scalability (e.g., eventual consistency).