BIG DATA ANALYTICS AND REASONING

INTRODUCTION

What do we mean by **big data**? We mean data whose **volume**, **variety**, **velocity** of production and **complexity** require new architectures, techniques, algorithms and **analytics** to manage it and extract **value** and **hidden knowledge.**

To define big data, we can use the famous "**Vs**":

- **Volume**: scale of data (just think at Google and the huge history of information).
- **Variety**: different forms of data, like text, images, etc.
- **Velocity**: speed of production and elaboration (for example, streaming data and logs); the internet produces continuous and intensive amounts of information.
- **Veracity**: uncertainty and imprecision of data; the quality of the data usually must meet some criteria but in this case it's not so necessary, all the data are mined and stored.
- **Value**: exploit intrinsic value by data to create business advantage (for example, obtaining preferences for ads) and as need for strong analytics and reasoning (Data Science).

From the historical point of view, there were three important revolutions which lead to the born of big data.

1. The emergence of the electronic computer.
   The emergence of electronic computers (the necessity to have pcs) following the Second World War ignited the first revolution in databases. Computers allow to store data in files and to direct high-speed access to records. The first-generation databases ran exclusively on the mainframe computers in which the emerged models were the Network model and the Hierarchical model. Unluckily, these kinds of databases were extremely inflexible because the queries must be anticipated during the design phase and complex analytic queries required complex coding; in addiction, all the databases were dependent from the specific application (customer-built).
2. The emergence of the relational database.
   Existing databases were too hard to use and mixed logical and physical implementations, so they could be used only by people with specialized programming skills. This inaccessibility leads to the definition of the first relational database model by Codd (1970). The relational database model clearly presents data to the user and does not mention how it should be stored on disk or in memory. The need to modify data simultaneously for multiple users led to the birth of transactions: a transaction is a transformation of state which has the properties of atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation); ACID transactions: atomic, consistent, isolated and durable. Transactions ensure consistency and integrity of data and became the standard for all serious database implementations. With the advent of the relational database, side minicomputers replaced mainframes and RDB started to ensure relational data model, ACID transactions and SQL (Structured Query Language).
   During the growth period of the relational data model, another paradigm started to be used and spread: the object-oriented programming. The OO programming involves two main concepts: the encapsulation, an object class encapsulates both data and actions that may be performed on that data, and the inheritance, the object classes can inherit the characteristics of a parent class. The spreading of this paradigm creates a new type of problem: the impedance mismatch, the representational techniques used in object-oriented programming representation do not match the relational database model, in which the data is represented in a table.
3. The need of global scope and continuous availability.
   A new era marked using massive web-scale applications begun and the relational model demonstrates scalability limits and high costs. In particular, Google had to invent new hardware and software architectures and Amazon, needing transactional processing capability that could operate at massive scale, had to think at new solutions. Between the new solutions there were new kinds of

databases, like the key-value store (DynamoDB) and the document databases (MongoDB), kinds of databases called NoSQL (start of the Nonrelational Explosion). Among the previous solutions, there were even relational ones that involves the process of sharding: partitioning the data across multiple databases losing the ACID transactions.
The Google solution, instead, involves not only the database aspect but also the architecture aspect; Google focused its work on data distribution, revealing a new distributed file system (GFS), a new parallel processing algorithm (MapReduce) and a new kind of database (BigTable).

GOODBYE SQL

As said earlier, at some point the usage of massive web-scale applications and the advent of the Web 2.0 with the three-layer architecture (client-server-database) pushed to the necessity of a new type of database architecture, far away from the original one. While until that moment the concept of scalability was connected to the vertical scalability (scale up), the adding of resources to a single node in a system, typically involving the addition of CPUs or memory to a single node, with the new revolution, a new type of scalability become used, the horizontal scalability (scale out), in which more nodes are added to a system.

The first attempt of implementing the horizontal scalability involved:

- adding a new type of machine, the Memcached servers: they avoid database access as much as possible and provide a distributed object cache to store object-oriented representation of database information across many servers.
- Implementing the concept of replication: it allows changes to one database to be copied to another database, creating a replica of the first one. The read requests can be directed to any one of these replica databases while the write operations still must go to the master database.
- Implementing the concept of sharding: a database is partitioned across multiple physical servers; in particular, the largest tables are partitioned horizontally (using Key value). So, with the word shard we mean a specific database partition. The application must determine which shard will contain the data and then send the SQL to the appropriate server. Sharding is simple in concept, but the application logic must understand the location of data; in addition, requests accessing more than one shard need complex coding.

Focusing on the sharding implementation, however, the operational costs are huge: not only complex code is necessary to maintain the system, but it's not even possible to issue a SQL statement that operates across shards and ensure the ACID transactions; furthermore, load balancing across shards is extremely problematic and adding new shards requires a complex rebalancing of data. For those reasons, many alternatives were developed among which the Oracle one: RAC, Real Application Cluster; RAC is a set of transparently scalable, ACID compliant, relational clusters in which each node is a database that works with data located on shared storage devices (shared disk clustering). Every node uses a distributed memory cache divided across databases node. However, this solution was too expensive to become effectively famous.

Using distributed solutions more and more often, we realized some limitations of these, which were then formalized in the CAP theorem: in a distributed system you have at most only two of Consistency (any user has an identical view of the system at any given instant), Availability (in the event of a failure, the system remains operational) and Partition tolerance (maintain operations in the event of the network failure).

CAP by example (1)

- This example is by Kaushik Sathupadi
  - Available at: http://ksat.me/a-plain-english-introduction-to-cap-theorem/

- **Chapter 1: "Remembrance Inc" Your new venture :**
  - Remembrance Inc! - Never forget, even without remembering!

    - Customer : Hey, Can you store my neighbor's birthday?
    - You: Sure.. when is it?
    - Customer : 2nd of jan
    - You: (write it down against the customer's page in your paper note book) Stored. Call us any time for knowing your neighbor's birthday again!
    - Customer : Thank you! You: No problem! We charged your credit card with $0.1

CAP by example (2)

- **Chapter 2 : You scale up:**
  - You and your wife both get an extension phone
  - Customers still dial (555)–55-REMEM and need to remember only one number
  - A pbx will route the a customers call to whoever is free and equally
- **Chapter 3 : You have your first "Bad Service"**
  - Jhon: Hey
  - You: Glad you called "Remembrance Inc!". What can I do for you?
  - Jhon: Can you tell me when is my flight to New Delhi?
  - You: Sure.. 1 sec sir
    (You look up your notebook)
    (wow! there is no entry for "flight date" in Jhon's page)!!!!!
  - You: Sir, I think there is a mistake. You never told us about your flight to delhi
  - Jhon: What! I just called you guys yesterday!(cuts the call!)

CAP by example (3)

- How did that happen?
  - Could John's call yesterday reached your wife?
  - Sure enough it's there.
  - **Your distributed system is not consistent!**
- **Chapter 4: You fix the Consistency problem:**
  - Whenever any one of us get a call for an update
    (when the customer wants us to remember something)
    before completing the call we tell the other person
  - This way both of us note down any updates
  - When there is call for search we don't need to talk with the other person
  - Since both of us have the latest updated information in both of our note books we can just refer to it.

CAP by example (4)

- This is sure, right?
  - BUT: an “update” request has to involve both and no work in parallel during write
  - **Availability problem!**
- **Chapter 5: You come up with the greatest solution Ever:**
  - "This is what we can do to be consistent and available"
    - i) Whenever any one of us get a call for an update before completing the call, if the other person is available we tell the other person. This way both of us note down any updates
    - ii) But if the other person is not available we send the other person an email about the update
    - iii) The next day when the other person comes to work after taking a day off, He first goes through all the emails, updates his note book accordingly.. before taking his first call

CAP by example (5)

- Consistent and available, right?
  - BUT: what if both of you report to work and one of you doesn’t update the other person?
    - You can decide to be partition tolerant by deciding not to take any calls until you patch up with your wife
  - You lost **Partition Tolerance!**
- **Chapter 7: Conclusion:**
  - You can get cannot achieve all three of Consistency, Availability and Partition tolerance.
    - Consistency: You customers, once they have updated information with you, will always get the most updated information when they call subsequently. No matter how quickly they call back
    - Availability: Remembrance Inc will always be available for calls until any one of you (you or your wife) report to work.
    - Partition Tolerance: Remembrance Inc will work even if there is a communication loss between you and your wife!

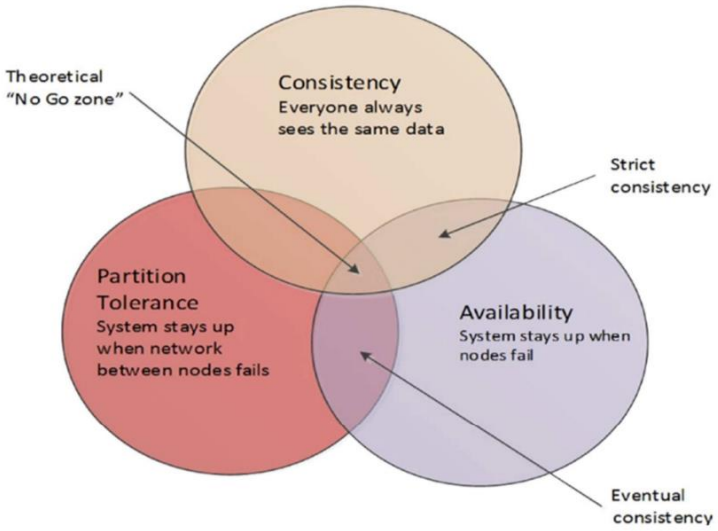
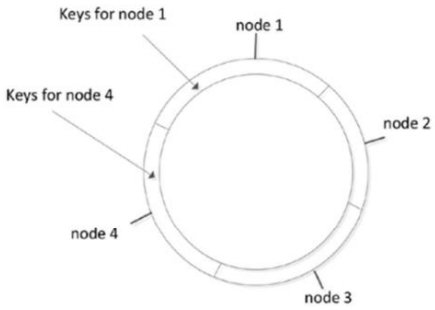


Figure 3-4. *Dynamo and ACID RDBMS mapped to CAP theorem limits*

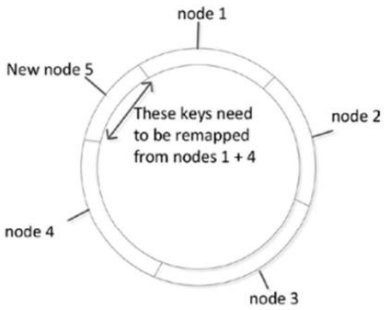
When we obtain availability and partition tolerance, sacrificing the consistency, we are working with eventual consistency: the updates happen in background, so there is some latency. An example of eventual consistency is the following one: does matter if my friend in Australia can see my tweet a few seconds before my friend in America? Eventual consistency, which relates more to performance than to availability, is possible because worldwide synchronous consistency is unnecessary.

One of the alternatives to the relational databases is Amazon's Dynamo, which exploit the eventual consistency to save the availability and the partition tolerance. Dynamo supports primary key-based access and the values retrieved by a key lookup are unstructured (so there is no structure on the payload) small binary objects. The key features of Dynamo are:

- Consistent hashing: the hash value of the primary key is used to determine the nodes in the cluster responsible for that key; with this method, nodes are added or removed from the cluster with minimal rebalancing overhead.

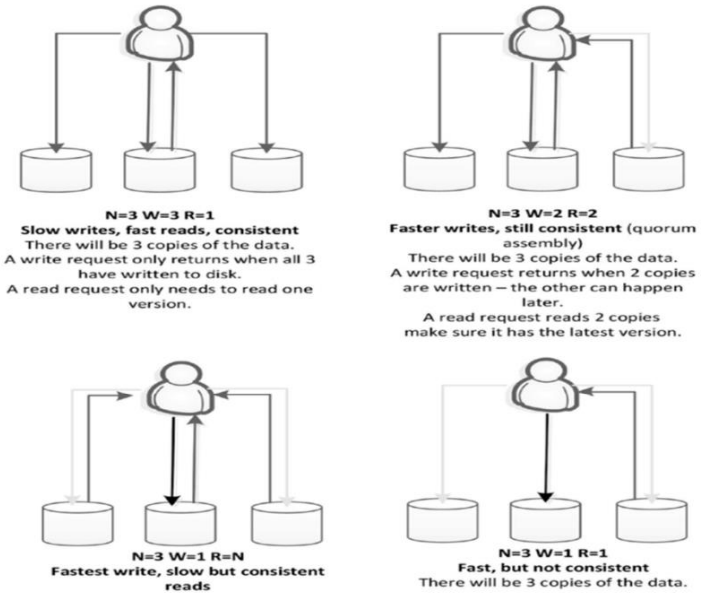


Split of keys in 4-node cluster



Adding a new node to the cluster

- Tunable consistency: the application can specify trade-offs between consistency, read performance and write performance.



- Data versioning: since write operations will never be blocked, it's possible that there will be multiple versions of an object in the system, so, these multiple versions need to be resolved by the application or users.

Another example of alternative to the relational database is the document database. A document database is a nonrelational database that stores data as structured documents, mostly in XML or JSON formats. Document databases are free to implement ACID transactions and are a medium between the rigid schema of the relational database and the completely schema-less key-value stores. The first document databases were XML-DBs.

- XML databases were not really positioned as alternative to the RDBMS, in fact, relational database vendors introduced XML support. Given the fact that XML tags are verbose and repetitious, and for this reason typically increasing the amount of storage (so, they are also computationally expensive to process), a new format started to spread, the JSON.
- The JavaScript Object Notation was mostly used as support to web-based operational workloads, and it was deliberately intended to be a lightweight substitute for XLM. JSON format became famous also because it removes the need of ORM (Object-Relational Mapping) in web applications. With its spreading, JSON became also famous as format for document databases.
    - In a JSON document database, a document is the basic unit of storage and corresponding approximately to a row in a RDBMS; a document comprises one or more key-value pairs and may also contain nested documents and array.
    - In JSON documents databases, a collection (or data bucket) is a set of documents (corresponding to a relational table) that doesn't have to be of the same type but represents a common category of information.
    - JSON documents databases are redundant to be efficient and it's like some join queries are materialized in the data itself.

[Between the early JSON databases, even Hadoop.]

GOOGLE STACK AND HADOOP

Google, one of the most important and famous search engines, was at first based on PageRank: given the fact that the World Wide Web was already huge, to order the big number of pages of the web, PageRank ranked pages depending on the number of links (so it used not only the content of the page but even its connections). To handle the different requests and the always bigger web, Google decided to develop a Google Modular Data Center: a hardware infrastructure capable of unbounded growth which included a very large numbers of commodity servers. However, no actual operating system or database platform could operate on a huge number of servers, so Google developed an in-house solution to massively parallelize and distribute processing, the Google Software Stack, which included all the protocols, the systems and the software to operate on the GMDC.

The Google Software Stack is composed by three major software layers:

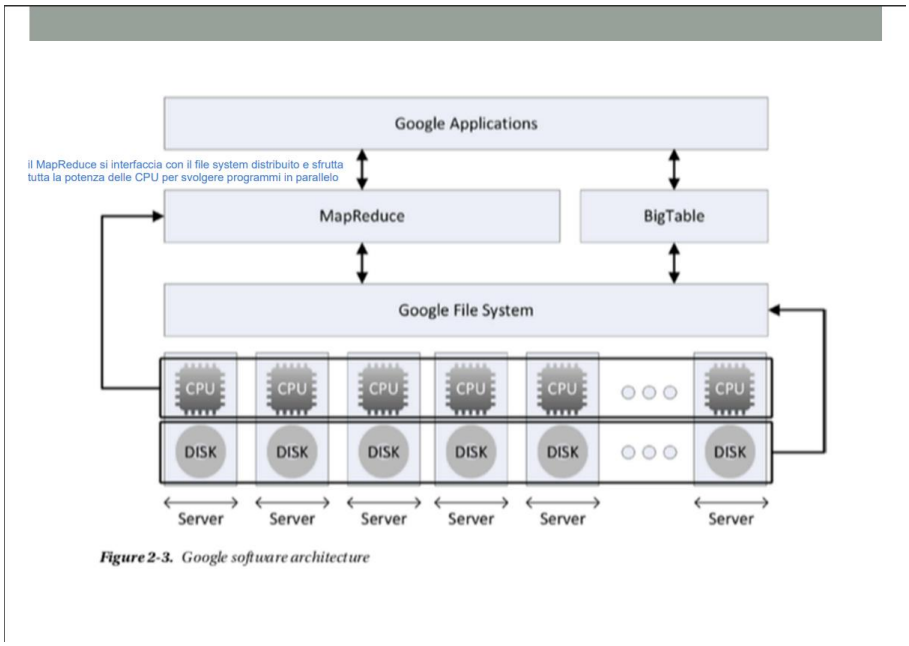
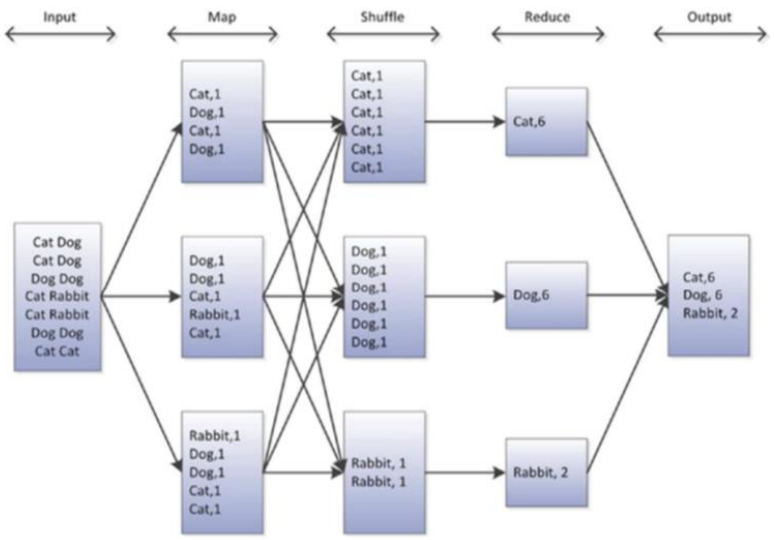


Figure 2-3. Google software architecture

- Google File System (GFS): a scalable distributed cluster file system for large, distributed data-intensive applications (applications which read a lot of data and need very fast processes to appending information) that allows all the disks accessed as one massive, distributed, redundant file system (a file is stored in pieces in different machines but from the outside it seems stored as a unique file). The goals of the file systems are performance, scalability, reliability, and availability; in particular, given the fact that the architecture is composed of commodity pcs, which are more inclined to damage, to obtain a higher scalability, some other key goals are:
    - Fault tolerance (recover promptly from component failures on a routine basis).
    - High sustained bandwidth is more important than low latency (in addition to being many commodity PCs, they can also be very far from each other, which is why bandwidth is important).
    - Atomicity with minimal synchronization overhead is essential.

  The file system was designed considering that modest number of large files should be managed efficiently; especially, large streaming reads and many large sequential writes were operated on the system (when working with Big Data we usually tend to do analysis, study the flow or the streaming of different files and therefore "random" accesses, such as obtain an ID etc. are less frequent).
  The cluster which will operate on the GFS consists of:
    - A single master: it maintains all file system metadata in main memory and do periodically the log for recovery. In addition, the master periodically communicates with each chunkserver to know if it is still operative.
    - Multiple chunkservers (the slaves): they store chunks (portion of a file) on local disk; each chunk is replicated on multiple chunkservers on different racks.
    - Multiple clients: they communicate with the master and the chunkservers; clients never read and write file data through the master, they ask the master which chunkservers they should contact, caches this information, and interact with the chunkservers directly for many subsequent operations.
- MapReduce: a distributed processing framework for parallelizing algorithms (or data-intensive processing) across large numbers of potentially unreliable servers. The framework includes two main phases:
    - Map: data is broken up into chunks that are processed in parallel.
    - Reduce: combines the output from the mappers into the final result.



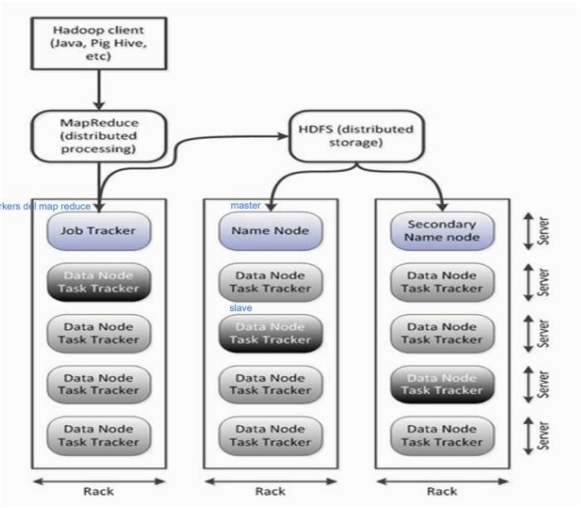
Remember that during the process, we don't work directly with the real object, but we work with copies, in this way we try to reduce times to resolve the waiting of the multithreading processes.

- BigTable: a nonrelational column-based database system that uses the Google File System for storage. Precisely, BigTable is a Sparse Distributed Persistent Multi-dimensional sorted map. The map is indexed by a row key, a column key and a timestamp (used to store in decreasing order different versions of the same information, to use the sharding) and each value in the map is an uninterpreted array of bytes (the data are stored as schema-less because we can't know in advance what kind of format the data will be stored): (row: string, column: string, time: int64) -> string.
  Even though the database is columnal, this does not mean that there is no concept of row. To store the row you need an identifier, in this case it's a URL, and then each row has a set of column families (which are many columns that have a common meaning). Each column has the associated value of the row.
  So, the row keys in a table are arbitrary strings and every read or write under a single row key is atomic. We can have row range (sets of strings), called tablet, for a table, each tablet is dynamically partitioned. Also, column keys can be grouped into sets called column families which are partitioned based on the row range; all data stored in a column family is usually of the same type, so access control and both disk and memory accounting are performed at the column-family level. Column family must be created before data can be stored.
  There are three major components of BigTable:
  - Client.
  - Master Server, which assigns/detects tablets in tablet servers (it has an index of all the tablets) and for this reason it balances tablet-server load.
  - Tablet server, which manages a set of tablets (the logic representation) and the relative read and write requests. Related to the tablet servers there is the Chubby, a distributed lock service which acts like a DHCP: it discovers tablet servers and finalize tablet servers' deaths; it ensures there is at most one active master and that clients communicate directly with tablet servers. The Chubby provides a namespace that consists of directories and small files where each directory or file can be used as a lock and reads and writes to a file are atomic.

At some point, anyways, Google's great power began to be envied and many wanted to exploit the same implementation which, however, was not open source; so, in 2007 an open-source version of Google's Software was published with the name of Hadoop. Hadoop can run on commodity hardware, like Google's software, and adding new servers to Hadoop adds storage, IO, CPU, and network capacity all at once. Given the large number of computers, data is stored redundantly in multiple servers distributed across multiple computers racks, so failure of a server does not result in a loss of data. Similarly, to Google, Hadoop takes inspiration from the nonrelational databases, so depending on the format of the data you are reading, you

will adapt your operations (no conversion to a normalized format): schema on read (the imposition of structure can be delayed until the data is accessed) vs schema on write.

This is the corresponding architecture from the Hadoop point of view.



Google File System	Hadoop Distributed File System
Map Reduce	Map Reduce (with Java)
BigTable	HBase (Hive – SQL for Hadoop)

Hadoop is different from the RDBMS because of:

1. It prefers to scale-out instead of scale-up.
2. It uses functional programming (Map Reduce) instead of declarative queries.
3. It implements offline batch processing (no real time) instead of online transactions: in Big Data we are more interested in analysis of big portion of data, so we prefer efficiency to real time answering.

The main architecture of Hadoop is composed as follows:

- NameNode: it is the master of the architecture; it knows how files are broken down into blocks and which nodes store those blocks (these are metadata).
- DataNode: the equivalent of the chunkserver, each slave machine in the cluster hosts a DataNode daemon. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.
- Secondary NameNode: it takes a snapshot of the HDFS metadata at defined intervals, and it is used as recovery node in case of NameNode failure.
- JobTracker: it determines the execution plan by determining which files to process and assigns to the node different tasks. After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assign different map and reduce tasks to each TaskTracker in the cluster.
- TaskTracker: it manages the execution of individual tasks on each slave node. There is a single TaskTracker per slave node which spawn multiple JVMs to handle many maps or reduce tasks in parallel.

GRAPH, COLUMN, AND IN MEMORY DATABASES

As we have already said, different kinds of databases started spreading to substitute the relational model. Now, we will list some of them.

GRAPH DATABASES

Databases usually store information about objects and their relationships and relationships have a natural representative counterpart, the graphs. Sometimes, relational model meets performance issues in the usage of graphs: even if, relational databases can easily model graphs, SQL might be a little limiting, especially in the depth searches: after the third join there are no optimizations implemented so each level of traversal of the graph adds significantly to query response time. Nevertheless, NoSQL are even worse at modeling graphs because relationships between objects are not inherently supported (no joins), the only one possibility is to store directly a graph. From this prospectives, a new kind of database born, the graph database.

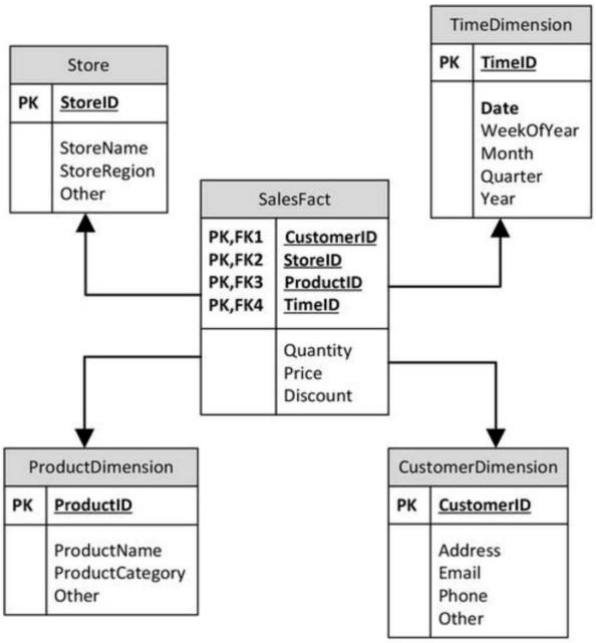
A graph is, obviously, given by a set of nodes N, a set of edges E, which modeling associations between two nodes, and N and E can have associated properties. Every query ends up in performing a graph traversal because the goal is to find a relation, sometimes exploring deeply the graph. One of the most widely adopted graph database is Neo4j in which both nodes and relationships are associated with attributes (property graph model).

Graph database can be very efficient because they don't need to use indexes to efficiently navigate the graph: each node already knows the physical location of all adjacent nodes; this enabling efficient real-time graph processing.
However, graph databases are not easy to distribute because of the overhead of routing the traversal across multiple machines and this eliminates the advantages of the graph database model; in fact, inter-server communication is far more time consuming than local access.
Anyways, graph processing, and other data models, work across massive, distributed datasets. Examples of graph databases which allow this are:

- Titan: a scalable graph database optimized for storing and querying graphs distributed across a multi-machine cluster.
- GraphX: part of the Berkeley Data Analytic Stack, it uses Spark as the foundation for graph processing.
- Apache Giraph: designed to run over Hadoop data using MapReduce.

TRIPLESTORES, COLUMN DATABASES AND STAR SCHEMA, C-STORE

- With the spreading of the internet, a new web standard born, the Resource Description Framework, which expresses information using triples, (entity: attribute: value). Together with this new format, even new native RDF databases born, the triplestores, which used the SPARQL query language.
- Up to that point, the most used databases were those oriented to OLTP processing (Online Transaction Processing) for which the CRUD operations were the most time-critical ones, so the row-based storage already provided good performance. With the advent of Big Data, data warehousing and analytic workloads led to a new time of processing, the OLAP processing (Online Analytic Processing). What are the differences between the two?
    - OLTP systems:
        - ACID transactions and CRUD operations.
        - Row-oriented physical organization ideal (write-intensive).
        - Service-level response times (immediate times that can guarantee the service).
    - OLAP systems:
        - IO intensive aggregate queries.
        - New non-normal-form schemas: Star & Snowflake.



Central large fact tables are associated with numerous smaller dimension tables. In this way, aggregate queries execute quickly.

- Column-oriented physical organization ideal (read-intensive). In column-oriented databases, data of a column are grouped together on disk, allowing advantage in aggregate queries where we are interested in data of the same column. However, retrieving a single row involves assembling the row from each of the column stores for that table.

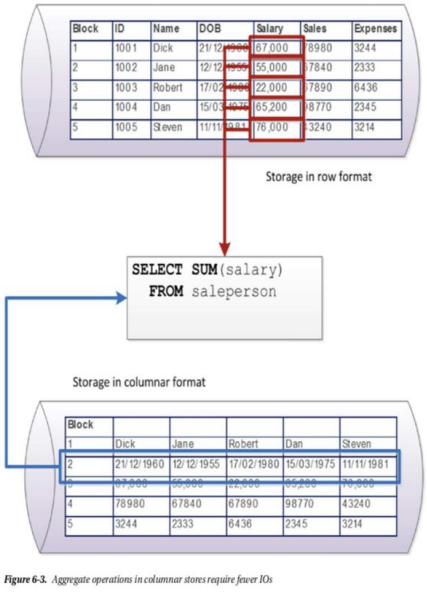


Figure 6-3. *Aggregate operations in columnar stores require fewer IOs*

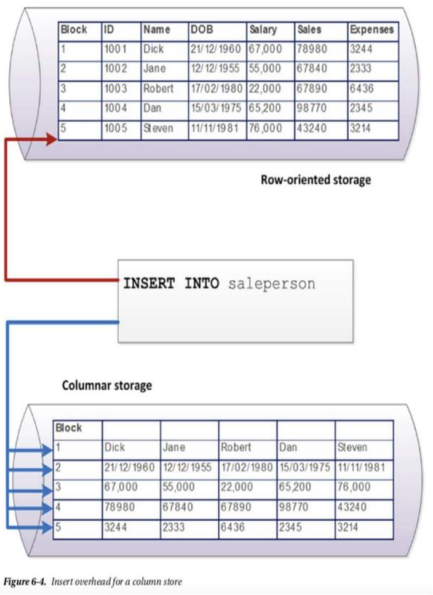
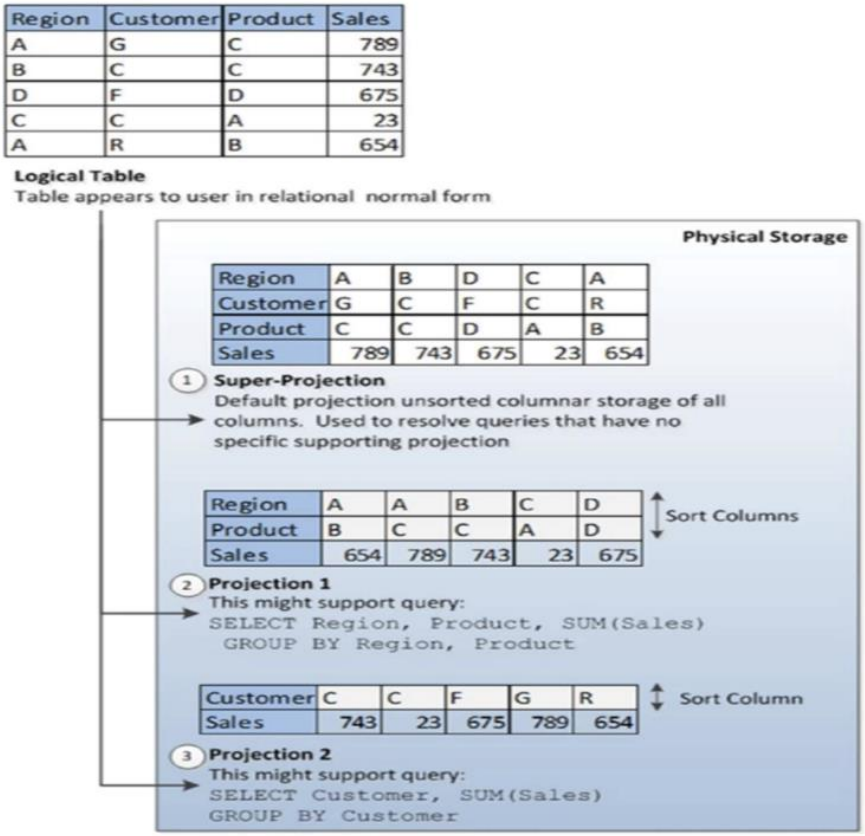
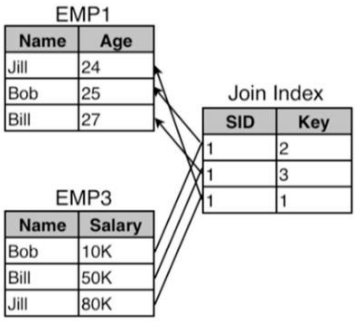


Figure 6-4. *Insert overhead for a column store*

- Both OLAP and OLTP system: C-STORE.
  C–STORE stores (possibly sorted) combinations of columns that are frequently accessed together on disk, even if redundant this is very efficient. The combination of columns is called projection, and a set of projections covers the entire table (so we end up with the original table); if something misses, additional projections are created to support specific queries.



Data are horizontally partitioned into segments that associate data values of every column with a storage key. To efficiently run queries, we use join indices that reconstruct all the records in a table.



Data is usually stored in sorted order, so we can decide to compress subsets of data (columns) together. Compressing data, we are exchanging CPU time for I/O bandwidth.
We say that C-STORE is a hybrid architecture because even if it is a columnar database and use column-oriented optimizer and executor, it tries to improve frequent insert and update and query performance. One of the attempts to improve performance is the use of k-different copies of the projection in different places.
The main difference of the C-STORE is the usage of the delta store: a in-memory (in RAM) row-oriented database which is used during the writing operation; at first the changes are written on the delta store and periodically the real database is updated based on the delta store. It's important to specify that queries can need both the store: if the main database is not updated, the reading operations are performed on the delta store.

Until that moment, the magnetic disk device has been a pervasive presence within digital computing, but, at some point, solid state device started to become competitive. Reading performances of SSD are superior to the ones of the magnetic disk, read operations require, in fact, only a single-page IO; however, SSD is significantly slower in writes because writing a page requires an erase and overwrite of a complete block. For this reason, traditional relational databases perform poorly on SSD. Some nonrelational systems, to use SSD, starting to avoid updating existing blocks and perform larger batched write operations (friendlier to solid state disk performance).

An example of SSD-oriented database is Aerospike: NoSQL database that has a log-structured file system in which updates are physically implemented by appending the new value to the file and marking the original data as invalid.

Similar to the C-STORE, there are a lot of databases which exploit RAM instead of disk, those databases are called in-memory databases (main memory DBs):

- Cache-less architectures: given the fact that the database is already in the memory, it's useless to use a cache for the data.
- Alternative persistence model: data in memory disappears when the power is turned off so alternative mechanism for ensuring no data loss are implemented; for example:
    - Replicating data to other members of a cluster.
    - Writing complete database images to disk files.
    - Writing out transaction/operation records to a transaction log or journal.
- TimesTen: SQL-based relational model, the data are always stored in the main memory and periodically the database takes snapshots of the memory and save the current state through a checkpoint in the disk (transaction log following a transaction commit).
- Redis: this is a particular key-value store in-memory. It can be configured to periodically save data or use a transfer log and supports data replication across multiple nodes.
- SAP HANA: this is a particular in-memory database which allows both row-oriented and column-oriented tables. In particular, the row tables are always in memory and read and write operations can be directly performed on the row tables. The column tables are loaded from a persistent layer only when they are needed and write operations are performed on a delta store like the one of the C-STORE.

BERKELEY ANALYTICS DATA STACK

Always talking about in-memory workflow, now we will present the corresponding in-memory distributed architecture (like Hadoop but an in-memory version).

- Spark: in-memory, distributed, fault-tolerant processing framework (in-memory alternative of MapReduce). It excels at tasks that cause bottlenecks on disk IO in MapReduce.
  In Spark, data are defined as resilient distributed dataset: collections of objects that are string or any other Java/Scala object type. The RDDs are immutable and are partitioned automatically across nodes of the cluster. The methods used on RDDs are implemented by DAG operations: Directed acyclic graph, a more sophisticated paradigm than MapReduce.
    - Spark is thought to manage data that won't fit entirely into main memory by paging (when something doesn't fit in RAM, they will be written on the disk).
    - Spark can read from or write to local or distributed file systems and usually RDDs are represented on disk as text files or JSON documents.

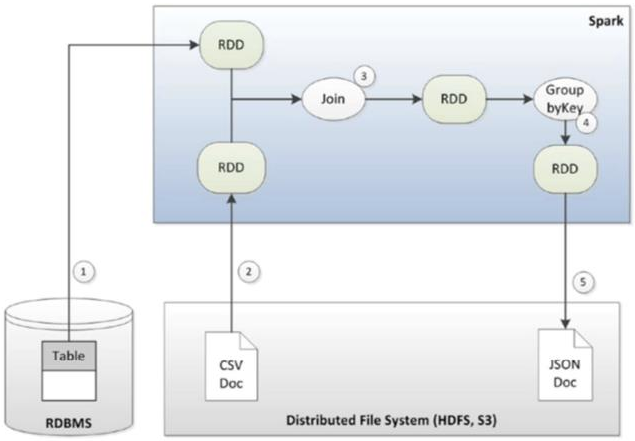


Figure 7-11. Elements of Spark processing

- Mesos: cluster management layer (in-memory alternative of Yarn). It allows the use of different frameworks.
- Tachyon: fault-tolerant, memory-centric distributed file system (in-memory alternative of HDFS).

All the following technologies are implemented to work together.

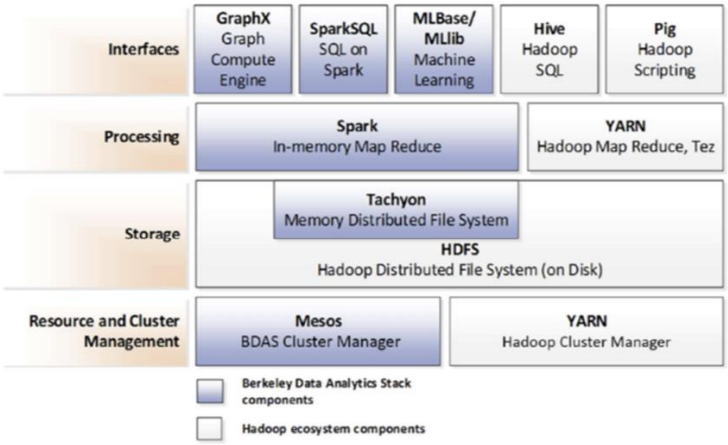


Figure 7-10. Spark, Hadoop, and the Berkeley Data Analytics Stack

MAP REDUCE - INTRODUCTION

Google needed to process large amounts of raw data, even if the computations are conceptually straightforward, they must be distributed across hundreds of thousands of machines. How to parallelize the computation, distribute the data, and handle failures without large amounts of complex code? Then, Google designed a new abstraction which goal was to make programmers without any experience with parallel and distributed systems able to easily utilize the resources of a large distributed system. The new abstraction, called MapReduce, is a programming model for processing and generating large data sets which enables automatic parallelization and distribution of large-scale computations. It takes care of:

- partitioning the input data (using a fine-grained partitioning which makes the input not seems partitioned).
- scheduling the program's execution in the cluster.
- handling machine failures.
- managing the required inter-machine communication.

The two main components of the interface are:

- map(k1,v1) -> list(k2,v2): preprocessing phase in which we map some information to some keys.
  Input: a key/value pairs.
  Output: a set of intermediate key/value pairs.
- reduce(k2,list(v2)) -> list(v2): it depends on the code of the developer.
  Input: intermediate values having the same intermediate key (the produced key-value pair are ordered and grouped).
  Output: a set of values.

Both these functions are provided by the user and the results computed by the functions are stored in local machines (for the map phase) and on GFS (for the reduce phase).

EXAMPLE – WORD COUNT

```
map(String key. String value): //the value string contains all the split
// key: document name
// value: document contents
       for each word w in value:
              EmitIntermediate(w, “1”);

reduce(String key, Iterator values): //the iterator is needed to handle lists of values that are
too large to fit in memory
// key: a word
// values: a list of counts
       int result = 0;
       for each v in values:
              result += ParseInt(v);
       Emit(AsString(result));
```

- The input data of the map function is partitioned into a set of M splits; each split can be processed in parallel by different machines.
- The key space is (hash-)partitioned into R pieces, e.g. the number of reducers.
- Keys and values can be of different types but must be serializable (in order to be sent in the network) and key must be comparable, otherwise it will be impossible to group and order it.

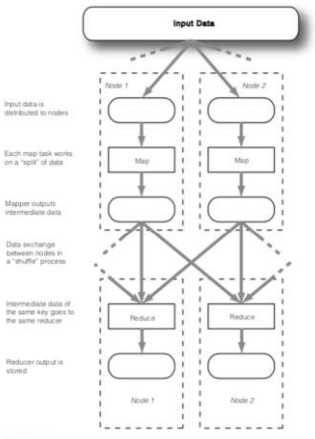
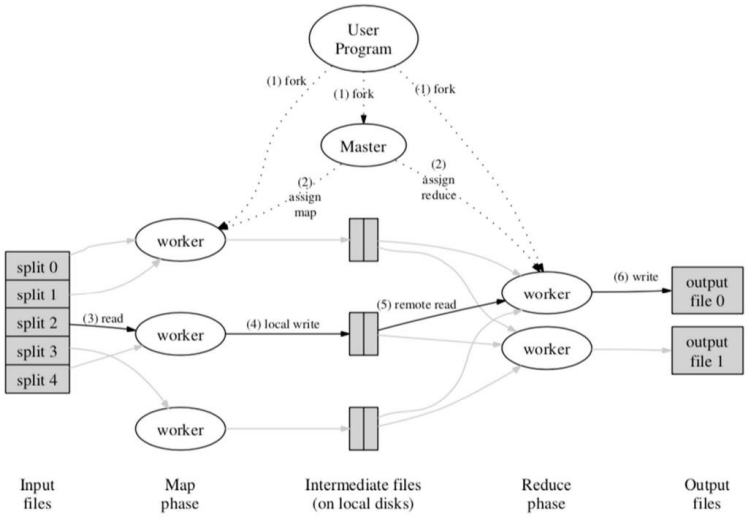


Figure 3.1 The general MapReduce data flow. Note that after distributing input data to different nodes, the only time nodes communicate with each other is at the "shuffle" step. This restriction on communication greatly helps scalability.

Steps of the MapReduce:

1. The library splits the input files into M pieces and starts up many copies of the program on a cluster of machines. One of the copies of the program is the master, the other copies are the workers to whom the job has been assigned by the master.
2. A worker who is assigned a map task reads the contents of the corresponding input split: it parses key-value pairs and calls the user-defined map function; then, the intermediate key-value pair are buffered in memory.
3. Periodically, the buffered pairs are written to local disk; the master is informed of the buffered pairs locations and forwards these locations to the reduce workers.
4. A reduce worker is notified by the master about locations and it uses remote procedure calls to read the buffered data from the local disks of the workers; then, it sorts data by the intermediate keys so that all occurrences of the same key are grouped together, in this way, many different keys can be mapped to the same reduce task.
5. The reduce worker iterates over the sorted intermediate data and calls reduce function for each unique intermediate key encountered. The output of the reduce function is appended to a final output file for this reduce partition.
6. When all map tasks and reduce tasks have been completed, the master wakes up the user program and the program returns back to the user code.



MapReduce is designed to handle machine failures; in particular, commits of map and reduce task outputs are made to achieve fault tolerance. The interface produces the same output as would have been produced by a non-faulting sequential execution for the entire program. There are two kinds of failures:

- Worker failure
  The master ping every worker periodically and if no response is received in a certain amount of time, the master marks the worker as failed and any map tasks completed by the worker are reset to idle state and become eligible for scheduling on other workers. Attention! Completed map tasks are re-executed because their output is lost in the local file system, but complete reduce tasks do not need to be re-executed because their output is stored in a global file system.
- Master failure
  Its failure is unlikely but, anyways, if the master fails, the entire MapReduce computation is aborted.

There are other important issues to discuss about MapReduce implementation:

- Locality: network bandwith is a relatively scarce resource so input data is stored and read locally, so it consumes no network bandwidth. However, GFS divide each file and stores several copies of each block (typically 3 copies) on different machines; then, the master attempts to schedule a map task on a machine that contains a replica of the corresponding input data or schedules a map task near a replica of that task's input data.

- Task Granularity: the number of reducers R is a small multiple of the number of worker machines; the number of map M and the number of reduce is much larger than the number of worker machines (to achieve fault tolerance).
- Backup Tasks: master schedules backup executions of in-progress tasks and the task is marked as completed whenever either the primary or the backup execution completes.
- Ordering Guarantees: intermediate key-value pairs are processed in increasing key order to generate a sorted output file per partition and support efficient random-access lookups by key.
- Skipping bad records: bugs in user code cause the map or reduce functions to crash deterministically on certain records, so the master must be resilient to failures.

DETAILS ABOUT IMPLEMENTATION

The implementation of the class necessary to run MapReduce must follow some rules:

- All the class must serialize and work with serialized key/value pairs: this is important because we need to transfer files through the network.
- Values used in the class must implement the Writable interface while keys must implement the WritableComparable interface: in particular, keys must be comparable because they will be sorted at the reduce stage. The primitive types must, then, be wrapped in some new classes like BooleanWritable, IntWritable, Text (for string) and so on.

Custom types

```
public class Edge implements WritableComparable<Edge>{

    private String departureNode;
    private String arrivalNode;

    public String getDepartureNode() { return departureNode;}

    @Override
    public void readFields(DataInput in) throws IOException {
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }

    @Override
    public int compareTo(Edge o) {
        return (departureNode.compareTo(o.departureNode) != 0)
            ? departureNode.compareTo(o.departureNode)
            : arrivalNode.compareTo(o.arrivalNode);
    }
}
```

1 Specify how to read data in

2 Specify how to write data out

3 Define ordering of data

- Mapper maps input key/value pairs to a set of intermediate key/value pairs.
  import org.apache.hadoop.mapreduce.Mapper
  public class Mapper<KEY_INPUT, VALUE_INPUT, KEY_OUTPUT, VALUE_OUTPUT>
  public void map(Object key, Text value, Context context)

Class	Description
IdentityMapper<K,V>	Implements Mapper<K,V,K,V> and maps inputs directly to outputs
InverseMapper<K,V>	Implements Mapper<K,V,V,K> and reverses the key/value pair
RegexMapper<K>	Implements Mapper<K,Text,Text,LongWritable> and generates a (match, 1) pair for every regular expression match
TokenCountMapper<K>	Implements Mapper<K,Text,Text,LongWritable> and generates a (token, 1) pair when the input value is tokenized

- Reducer reduces a set of intermediate values which share a key to a smaller set of values; to do that, there are two main phases:

- Shuffle & sort: the reducer copies the sorted output from each Mapper using HTTP across the network and merge sort Reducer inputs by keys.
- SecondarySort: the reducer extends the key with the secondary key and define a grouping comparator.

import org.apache.hadoop.mapreduce.Reducer
public class Reducer<KEY_INPUT, VALUE_INPUT, KEY_OUTPUT, VALUE_OUTPUT>
public void reduce(Text key, Iterable<IntWritable> values, Context context)

Class	Description
IdentityReducer<K,V>	Implements Reducer<K,V,K,V> and maps inputs directly to outputs
LongSumReducer<K>	Implements Reducer<K,LongWritable,K,LongWritable> and determines the sum of all values corresponding to the given key

- Combiner: we now introduce a new module. The combiner performs a "local reduce" before we distribute the mapper results; this can be very useful to reduce network traffic and increase performance and to increase load balancing and avoid overwhelming a single reducer of a huge amount of data. The combiner implements the same interface of the Reducer, but its output is usually the same of the mapper. It is now always applicable.
- Partitioner: it determines where to send a key/value pair outputted by a mapper and controls the partitioning of the keys of the intermediate map-outputs (the total number of partitions is the same as the number of reduce tasks for the job). A partitioner is created only when there are multiple reducers
  public abstract class Partitioner<KEY,VALUE>
- It's important to define the way an input file is split up and read by Hadoop, this is possible using the InputFormat interface. Usually, we use also the InputFormat class which describes and validates the input-specification for a MapReduce job. It also splits-up the input files into logical InputSplits, each of which is then assigned to an individual Mapper.
  public abstract class InputFormat<K,V>

InputFormat	Description
TextInputFormat	Each line in the text files is a record. Key is the byte offset of the line, and value is the content of the line. key: LongWritable value: Text
KeyValueTextInputFormat	Each line in the text files is a record. The first separator character divides each line. Everything before the separator is the key, and everything after is the value. The separator is set by the key.value.separator.in.input. line property, and the default is the tab (\t) character. key: Text value: Text
SequenceFileInputFormat<K,V>	An InputFormat for reading in *sequence files*. Key and value are user defined. Sequence file is a Hadoop-specific compressed binary file format. It's optimized for passing data between the output of one MapReduce job to the input of some other MapReduce job. key: K (user defined) value: V (user defined)
NLineInputFormat	Same as TextInputFormat, but each split is guaranteed to have exactly *N* lines. The mapred.line.input.format. linespermap property, which defaults to one, sets *N*. key: LongWritable value: Text

Remember, if you want to implement a custom InputFormat, it must identify all the files used as input data and divide them into input slits (each map task, then, is assigned one split) and provide an

object (RecordReader) to iterate through records in a given split and to parse each record into key and value of predefined types.

- MapReduce output files using the OutputFormat class which is analogous to the InputFormat (even if the output has no splits). Each reducer writes its output only to its own file in a common directory in the distributed file system and typically named that file part-nnnnn, where nnnnn is the ID of the reducer.

OutputFormat	Description
TextOutputFormat<K,V>	Writes each record as a line of text. Keys and values are written as strings and separated by a tab (\t) character, which can be changed in the mapred. textoutputformat.separator property.
SequenceFileOutputFormat<K,V>	Writes the key/value pairs in Hadoop's proprietary sequence file format. Works in conjunction with SequenceFileInputFormat.
NullOutputFormat<K,V>	Outputs nothing.

- To access to configuration parameters, we can use the Configuration class.
  public class Configuration extends Object implement iterable<Map.Entry<String,String>>, Writable
  Configurations are specified by resources which contains a set of name/value pairs as XML data.
  Hadoop by default specifies two resources:
  - core-default.xml: read-only defaults for Hadoop.
  - core-site.xml: site-specific configuration for a given Hadoop installation.
- Once all the code is ready, the user can create the application and describe the details of the MapReduce job via the class Job; after they can submit the job to the cluster (so the master will take into charge) and monitor its progress.
  public class Job extends JobContextImpl implements JobContext, AutoCloseable
  With an instance of Job, it's possible to configure job (set the name, the jair containing the code, the mappers and the reducers, the input and output format and the main to run) and submit and control the execution using the job.waitForCompletion(true).

```
// Create a new Job
Job job = Job.getInstance();
job.setJarByClass(MyJob.class);

// Specify various job-specific parameters
job.setJobName("myjob");

job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));

job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);

// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

- Useful code to include Hadoop and MapReduce.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>it.unical.mat.bigdata</groupId>
<artifactId>Examples</artifactId>
<version>0.0.1-SNAPSHOT</version>

<dependencies>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-mapreduce-client-core</artifactId>
<version>3.2.0</version>
</dependency>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-common</artifactId>
<version>3.2.0</version>
</dependency>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-mapreduce-client-jobclient</artifactId>
<version>3.2.0</version>
</dependency>
</dependencies>
</project>
```

HADOOP CON ALTRI LINGUAGGI

The MapReduce framework can be used even with other languages different from Java but it's necessary to use the generic API called Streaming. This API interacts with programs using the Unix streaming paradigms; in particular, inputs come in through STDIN and outputs go to STDOUT, data must be text based and each line is considered a record.

```
bin/hadoop jar [hadoop-streaming-X.Y.Z.jar]
       -input [input_file]
       -output [output_file]     //cartella nel quale salvare l'output solitamente
       -file [script to share]
       -mapper 'mappercommand'
       -reducer 'reducercommand'
```

```
Usage: $HADOOP_HOME/bin/hadoop jar hadoop-streaming.jar [options]
Options:
  -input          <path> DFS input file(s) for the Map step.
  -output         <path> DFS output directory for the Reduce step.
  -mapper         <cmd|JavaClassName> Optional. Command to be run as mapper.
  -combiner       <cmd|JavaClassName> Optional. Command to be run as combiner.
  -reducer        <cmd|JavaClassName> Optional. Command to be run as reducer.
  -file           <file> Optional. File/dir to be shipped in the Job jar file.
                  Deprecated. Use generic option "-files" instead.
  -inputformat    <TextInputFormat(default)|SequenceFileAsTextInputFormat|JavaClassName>
                  Optional. The input format class.
  -outputformat   <TextOutputFormat(default)|JavaClassName>
                  Optional. The output format class.
  -partitioner    <JavaClassName>  Optional. The partitioner class.
  -numReduceTasks <num> Optional. Number of reduce tasks.
  -inputreader    <spec> Optional. Input recordreader spec.
  -cmdenv         <n>=<v> Optional. Pass env.var to streaming commands.
  -mapdebug       <cmd> Optional. To run this script when a map task fails.
  -reducedebug    <cmd> Optional. To run this script when a reduce task fails.
  -io             <identifier> Optional. Format to use for input to and output
                  from mapper/reducer commands
  -lazyOutput     Optional. Lazily create Output.
  -background     Optional. Submit the job and don't wait till it completes.
  -verbose        Optional. Print verbose output.
  -info           Optional. Print detailed usage.
  -help           Optional. Print help message.
```

In principle, even if the API works with the STDIN and STOUT, the input is read from a file and the output is redirected to a file.

- Each mapper sees the entire stream of data and takes on the responsibility of breaking the stream (in which the tab character separates the key from the value) into record of key-values pairs to pass them to the MapReduce framework. This is one of the differences between the java framework and the streaming API, usually it's the framework to break the input data while now it's the mapper itself.
- Then, the Streaming API will just do the essential operations to handle mapper's output and assign the keys to the reducers.

Example:

```
./bin/hadoop jar hadoop-streaming-X.Y.Z.jar
     -input imput.txt
     -output out.txt
     -mapper 'cut -f1,3'
     -reducer 'maxGroupAttr.py'

# Equivalent to

cat imput.txt | cut -f1,3
     | sort | ./maxGroupAttr.py
```

maxGroupAttr.py

```
#!/usr/bin/python
import sys

lastKey=None
max   = 0
for line in sys.stdin:
        (key, val) = line.strip().split()
        if lastKey and lastKey!=key:
                print lastKey + "\t" + max
                max=0
                lastKey=None
        lastKey=key
        if (val > max):
                max = val
print lastKey + "\t" + max
```

- The Streaming API offers a library package called Aggregate that simplifies obtaining aggregate statistics of a data set. When this library is used, each line of the mapper's output looks like function: `key\tvalue`. It's important to remember that it's a task of the developer to specify which function to use. The aggregate reducer, instead, applies the function to the set of values for each key.

Value aggregator	Description
DoubleValueSum	Sums up a sequence of double values.
LongValueMax	Finds the maximum of a sequence of long values.
LongValueMin	Finds the minimum of a sequence of long values.
LongValueSum	Sums up a sequence of long values.
StringValueMax	Finds the lexicographical maximum of a sequence of string values.
StringValueMin	Finds the lexicographical minimum of a sequence of string values.
UniqValueCount	Finds the number of unique values (for each key).
ValueHistogram	Finds the count, minimum, median, maximum, average, and standard deviation of each value. (See text for further explanation.)

CHAINING MAPREDUCE JOBS

Sometimes, complex tasks need to be broken down and to be accomplished by different MapReduce job- Chaining MapReduce jobs in a sequence is useful and directly implementable using java: we can just schedule every next job after the one in the main and wait until preceding jobs are completed.

```
job.waitForCompletition(true) ? 0 : 1;
```

Complex chains are implemented using the ChainMapper class:

```
ChainMapper.addMapper(job, map1.class, key.class, val.class);
ChainMapper.addMapper(…);
ChainMapper.setReducer(job, red1.class, key.class, val.class);
…
job.waitForCompletition(true) ? 0 : 1;
```

We have to remember that all those mapper and reducer are disconnected between them so we can also add 4 mapper and 1 reducer or viceversa.

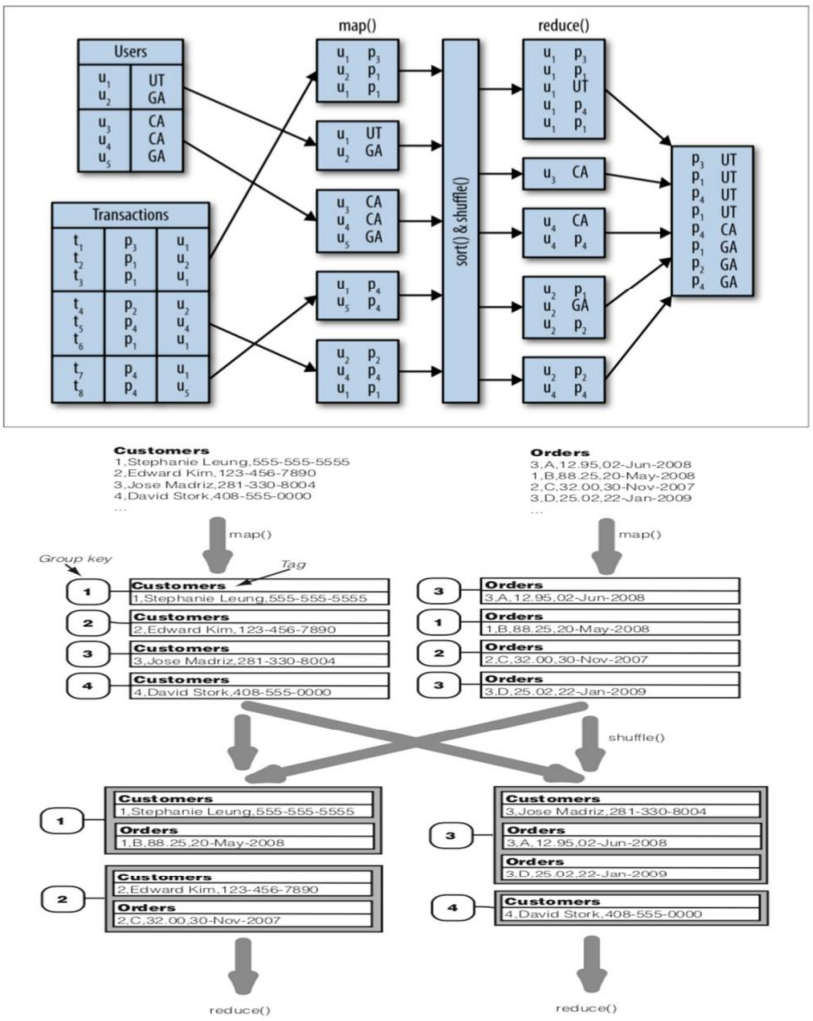
However, usually, we want to enforce specific dependencies, for example, a job1 must be computed after a job2 and so on.

```
ControlledJob cJ1 = new ControlledJob(job1);
ControlledJob cJ2 = new ControlledJob(job2);
cJ1.addDependingJob(cJ2); // so cJ2 depends on cJ1 and it will start after that
JobControl jc = new JobControl(“A chain”);
jc.addJob(cJ1); //every job must be add to the jobcontrol that based on the
jc.addJob(cJ2); // dependencies will start the right one
jc.run();
```

MAP REDUCE - JOINING DATA

How to implement join operations in Map Reduce? There are different possible patterns.

- Reduce-side join: also called repartitioned sort-merge join; it is the most general join technique, but it is not very efficient.
    1. The mappers set the group (join) key as key of the output, select relevant columns and emit them as values. The key to the pair is used to tag the data to mentioning the origin.
    2. The framework will shuffle and sort using the output mappers’ key.
    3. The reducers combine grouped values to generate the output so they actually decide what to preserve in the output tuple and perform the actual join.



The negative side of this kind of pattern are the following:

- Joining doesn't take place until the reduce phase.
- It's necessary to transfer all data across the network.
- Even If all the data are transferred, often the reducers drop most of this data.

- Map-side join: it removes the unnecessary data right in the map phase but it's important to highlight that mappers only see the associated split: a record being processed by a mapper may be joined with a record not easily accessible (or even located) by that mapper; so, we must guarantee the accessibility of all the necessary data to a mapper. This latter is possible only in two situations:
  - If the sources are physically organized in a sorted way, for example, sorted on the key that is the desired join key, so that each mapper can deterministically locate and retrieve all the data necessary to perform joining.
  - If one of the two sources is small enough, this because we must share one of the sources before starting the join operations. So, we need to have a smaller source that can fit in memory of the mappers: we copy the smaller source to all mappers and perform join in the map phase.

  To share the data, Hadoop has a mechanism called distributed cache that is designed to distribute files to all nodes in a cluster. So, the smaller table is added to the cache and when the mappers start their computation, they get the table from the distributed cache.
- Reduce-side join with map-side filtering: this pattern combines the map-side and the reduce-side techniques using semi-join (before to join the tables, the records are filtering using a check).
  - Mappers perform an approximated semi-join to filter out useless data and avoid as much as possible transmitting tuples that will be discarded.

- Reducers perform the usual join but on less data.

To filter the record, this pattern exploits the bloom filters. A bloom filter works as a probabilistic set: we add elements to the set and based on the previous elements added we decide if adding some elements or not. The more elements are added to the set, the larger the probability of false positives, the larger the structure of the filter, the smaller the error.

- Let B be a bit array of size m (m > 1), initialized with 0s.
- Let {H1, H2, …, Hk} be a set of k hash functions, if Hi(xj) = a, then set B[a] = 1.
- To check if x belongs to S, check B all k values Hi(x) must be 1.

```
Array B:
   initialized:
         index  0  1  2  3  4  5  6  7  8  9
         value  0  0  0  0  0  0  0  0  0  0

   insert element a, H(a) = (2, 5, 6)
         index  0  1  2  3  4  5  6  7  8  9
         value  0  0  1  0  0  1  1  0  0  0

   insert element b, H(b) = (1, 5, 8)
         index  0  1  2  3  4  5  6  7  8  9
         value  0  1  1  0  0  1  1  0  1  0

   query element c
   H(c) = (5, 8, 9) => c is not a member (since B[9]=0)

   query element d
   H(d) = (2, 5, 8) => d is a member (False positive)

   query element e
   H(e) = (1, 2, 6) => e is a member (False positive)

   query element f
   H(f) = (2, 5, 6) => f is a member (Positive)
```

- The probability of false positives is:

$$\left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}$$

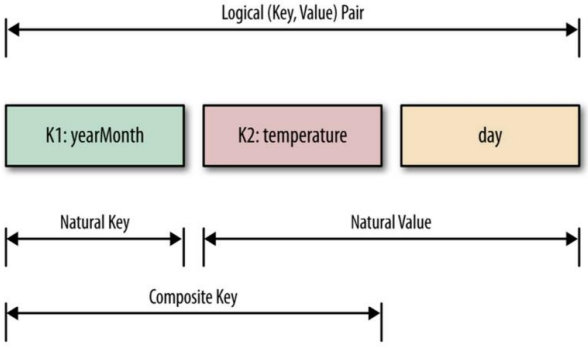
- What is the optimal number of hash functions?
- Given m (number of bits selected for Bloom filter) and n (size of big data set), the value of k (number of hash functions) that minimizes the probability of false positives

$$k=\frac{m}{n}ln(2) \qquad m=-\frac{nln(p)}{(ln(2))^2}$$

- Probability of getting 1 increases steadily with elements

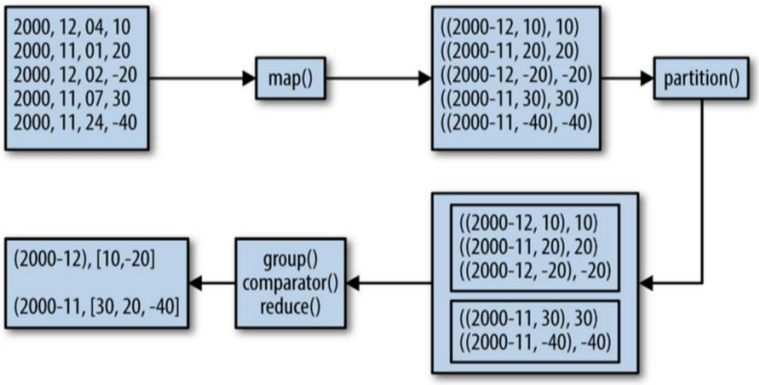
$$1-\left(1-\frac{1}{m}\right)^{kn} \approx 1-e^{-\frac{kn}{m}}$$

- Secondary sort: useful to implement joins with group by, it sorts the values passed to each reducer. The key idea is to offload the sorting to the MapReduce framework: what we have to do is creating a composite key (natural key + some value), provide custom comparators and partitioners (to partition the mapper's output by the natural key), and exploit map-reduce frameworks distributed sort (instead of collecting all values and sorting them in memory).



1. The mapper create (K,V) pairs, where K is a composite key.

2. The partitioner class enables sends all natural keys to the same reducer.
3. The grouping comparator class enables temperatures to arrive sorted at reducers.



At the implementation side, what we need is to:

- Create a specific datatype for the composite key using WritableComparable and Writable.
- Create a custom partitioner which extends Partitioner and it's added using `job.setPartitionerClass`.
- Create a grouping comparator, which controls which keys are grouped together for a single reduce() call, which extends WritableComparator and it's added using `job.setGroupingComparatorClass`.

MONOIDS – COMBINER

When too much work is required by the shuffle and sort of the framework, we would like to implement a better code to make the algorithm efficient; however, sometimes, we cannot use the reducer as a combiner. Then, when can we use the combiner? When the output has a precise algebraic property, so it allows us to perform a local reduce on the output of the mapper. The output of the mapper must be a monoid.

A monoid is a triple (S,f,e):

- S is a set.
- f: S x S -> S is a binary operation (for us *).
- e belongs to S and it is the identity element.

Closure: for all the elements a and b in S, the result of a * b is also in S.
Associativity: for all a, b and c in S, it holds that (a*b)*c = a*(b*c). [this property is very exploited in combiner]
Identity element: for all a in S, e*a = a*e = a.

- Addition over set of integers
  - 1+0=0+1=1
  - a+(b+c) = (a+b)+c
  - a+b is a number
- Maximum over Set of Integers → Monoid
  - MAX(a, MAX(b,c)) = MAX(MAX(a,b),c)
  - MAX(a,0) = MAX(0,a) = a
  - MAX(a,b) is a number
- Subtraction over Set of Integers → NOT a Monoid
  - (1-2) -3 ≠ 1-(2-3)
  - Not associative!

Then, we can use combiner when the function we want to apply is a commutative monoid (the binary operation is also commutative, then, for all a and b in S, it holds (a*b) = (b*a)).
In particular, we refer to commutative monoid because we know that there are some operations that don't need a particular order between data to be applied, then operating on commutative monoid is not a problem, because we are not taking into account the order.

At this point we are asking to ourselves, why monoids? Because monoids can build fold operations, e.g. operations that collapse a sequence of other operations into a single value; in this way, we can get a much less larger number of data to send into the network.

"Monoidify! One principle for designing efficient MapReduce algorithms can be precisely articulated as follows: create a monoid out of the intermediate value emitted by the mapper. Once we monoidify the object, proper use of combiners and the in-mapper combining techniques becomes straightforward."