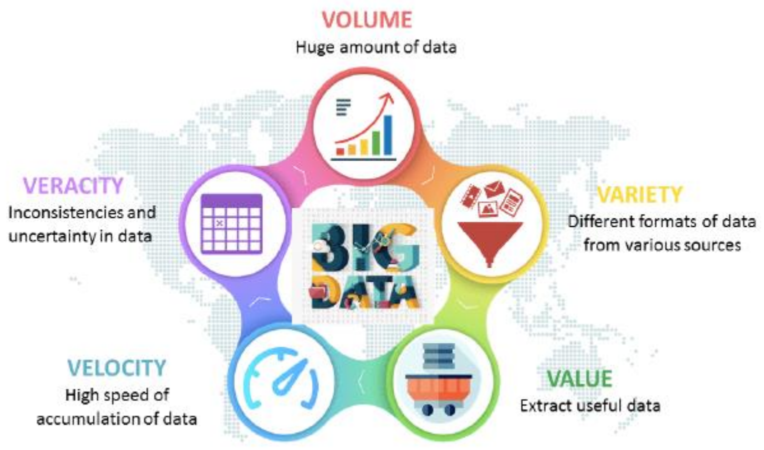
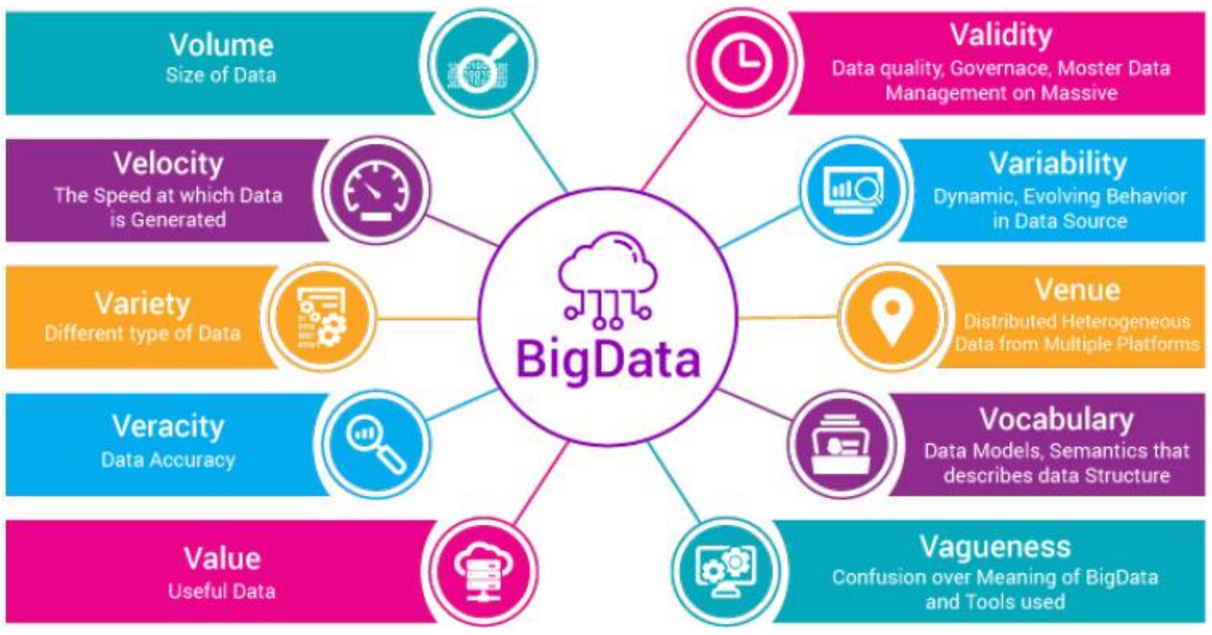
ACID: Atomicity, Consistency, Isolation, Durability

BASE: Basically Available, Soft State, Eventually Consistent

BIG DATA


VOLUME
Huge amount of data
VERACITY
Inconsistencies and uncertainty in data
BIG DATA
VARIETY
Different formats of data from various sources
VELOCITY
High speed of accumulation of data
VALUE
Extract useful data


Volume
Size of Data
Velocity
The Speed at which Data is Generated
Variety
Different type of Data
Veracity
Data Accuracy
Value
Useful Data
BigData
Validity
Data quality, Governace, Moster Data Management on Massive
Variability
Dynamic, Evolving Behavior in Data Source
Venue
Distributed Heterogeneous Data from Multiple Platforms
Vocabulary
Data Models, Semantics that describes data Structure
Vagueness
Confusion over Meaning of BigData and Tools used

Big Data refers to any problem characteristic that represents a **challenge** to process it with **traditional applications**.

Big Data involves **data** whose volume, diversity and complexity requires **new techniques**, **algorithms and analyses** to fetch, to store and to elaborate them.

Main Big Data Computing Issue:

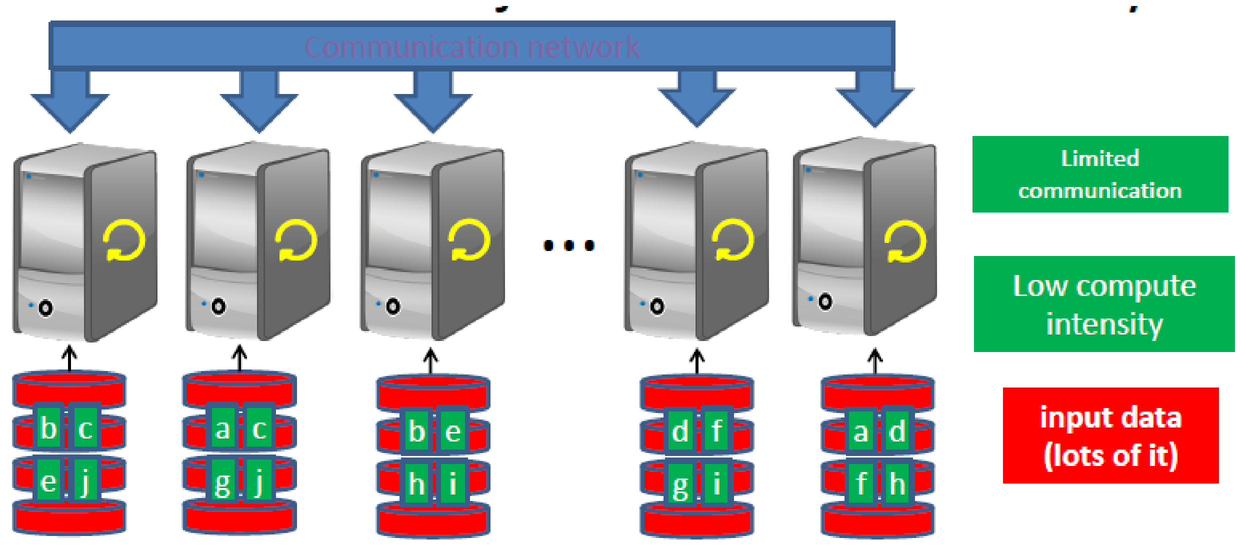
- **Classical** data elaboration algorithms cannot be directly applied to Big Data
- **New** technologies are needed for both **storage** and **computational** tasks
- **New paradigms** are needed for **designing, implementing** and **experimenting** algorithms for handling big data

To deal with data intensive applications we have the dighotomy **Scale-up** vs **Scale-out**.

One machine cannot process or store all data:

- Daya is **distributed** in a cluster of computing nodes
- It does **not matter** which machine executes the operation
- It does **not matteri f** it is **run twice** in different nodes
- We look for an **abstraction of the complexity** behind distributed systems

Data Locality is crucial; avoid data transfers between machines as much as possible.



Solution: store data on local disks of the nodes that perform computations on that data ("**data locality**")

MapReduce: moving computation is cheaper than moving computation and data at the same time.

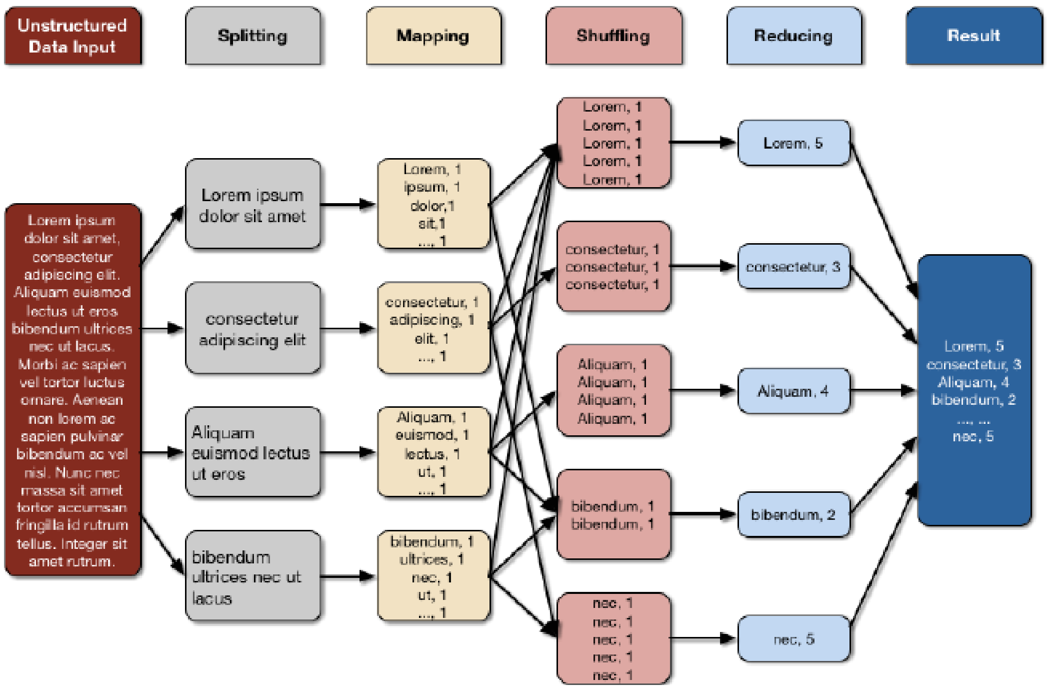
- **Data** is **distributed** among nodes.

- **Functions/operation**s to process data are **distributed** to all the computing nodes.
- Each computing node works with the data stored in it.
- **Only the necessary data is moved across the network**.

Divide & Conquer strategy:

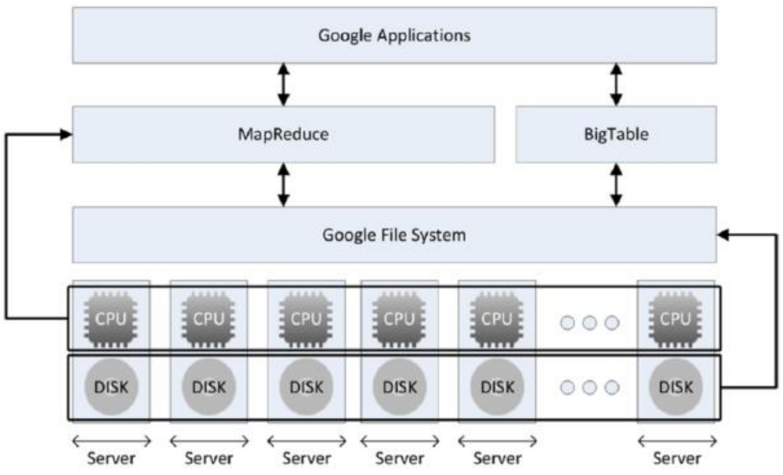
divide – partition dataset into smaller, independent chunks to be processed in parallel (*map*)

conquer – combine, merge pr ptherwise aggregate the results from the previous step (*reduce*)



Unstructured Data Input
Splitting
Mapping
Shuffling
Reducing
Result
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam euismod lectus ut eros bibendum ultrices nec ut lacus. Morbi ac sapien vel tortor luctus ornare. Aenean non lorem ac sapien pulvinar bibendum ac vel nisl. Nunc nec massa sit amet tortor accumsan fringilla id rutrum tellus. Integer sit amet rutrum.
Lorem ipsum dolor sit amet
consectetur adipiscing elit
Aliquam euismod lectus ut eros
bibendum ultrices nec ut lacus
Lorem, 1 ipsum, 1 dolor, 1 sit, 1 ...., 1
consectetur, 1 adipiscing, 1 elit, 1 ...., 1
Aliquam, 1 euismod, 1 lectus, 1 ut, 1 ...., 1
bibendum, 1 ultrices, 1 nec, 1 ut, 1 ...., 1
Lorem, 1 Lorem, 1 Lorem, 1 Lorem, 1 Lorem, 1
consectetur, 1 consectetur, 1 consectetur, 1
Aliquam, 1 Aliquam, 1 Aliquam, 1 Aliquam, 1
bibendum, 1 bibendum, 1
nec, 1 nec, 1 nec, 1 nec, 1 nec, 1
Lorem, 5
consectetur, 3
Aliquam, 4
bibendum, 2
nec, 5
Lorem, 5 consectetur, 3 Aliquam, 4 bibendum, 2 ...., ... nec, 5

Google Software Architecture for Big Data



This is the architecture **Google** used at the beginning (2000s).

It exploited the horizzontal scalability (put together a huge amount of low cost servers).

Made use of **Google File System** abstraction to represent data on disk as an abstraction. On this data work **MapReduce** computing paradigms and **BigTable** NoSQL database for queries.

At the upper level we have the **Google Application**, which works on huge amount data.

Hadoop

Hadoop is:
- An **open-source** (the first) framework written in Java
- Distributed storage of very large data sets (Big Data)
- Distributed processing of very large data sets

This framework consists of a **number of modules**
- ***Hadoop Common****: the common utilities that support the other Hadoop modules.*
- ***Hadoop Distributed File System (HDFS)***
- **Hadoop YARN** – A framework for job scheduling and cluster resource management
- ***Hadoop MapReduce*** – programming model

HDFS – **Hadoop Distributed File System**
- Distributed File System written in Java
- Scales to clusters with **thousands of computing nodes**
  • Each node stores part of the data in the system
- **Fault tolerant** due to data replication
- Designed for big files and low-cost hardware
  • GBs, TBs, PBs
- **Efficient for read and append operations** (random updates are rare)

Hadoop Limits:

• Hadoop is optimized for ***one-pass batch*** processing of on-disk data
• It suffers for ***interactive data exploration*** and more complex multi-pass analytics algorithms
• Due to a ***poor inter-communication*** capability and inadequacy for ***in-memory computation*** , Hadoop ***is not suitable*** for those applications that require ***iterative and/or online computation***

Spark

Apache Spark is an open-source which has emerged as the next generation big data processing tool due to its enhanced ***flexibility*** and ***efficiency***.

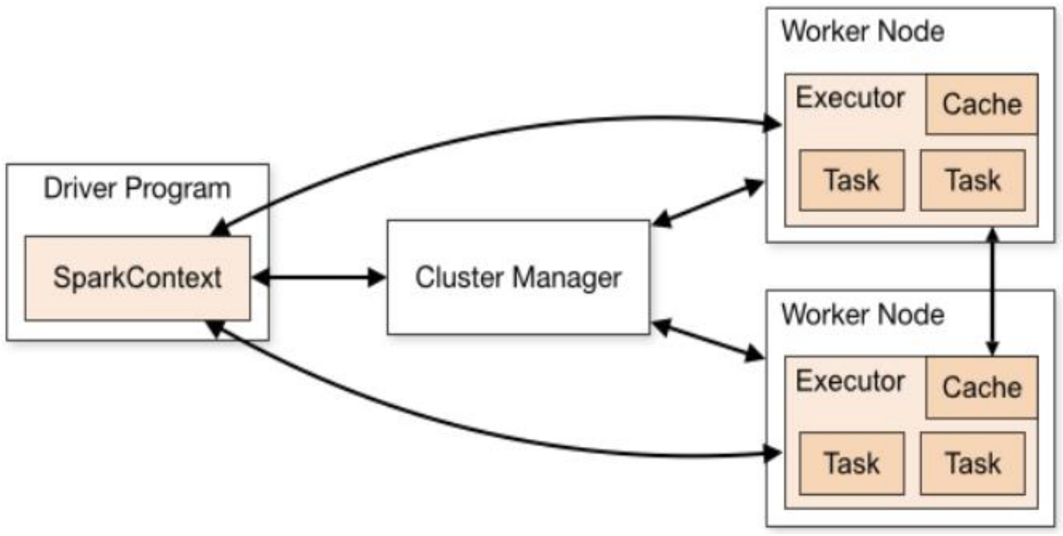
Spark allows employing ***different distributed programming models***, such as ***MapReduce*** and ***Pregel***, and has proved to perform ***faster*** than Hadoop , especially in case of ***iterative and online applications***.

Unlike the disk-based MapReduce paradigm supported by Hadoop, Spark employs the concept of ***in-memory cluster computing***, where datasets are cached in memory to reduce their access latency.

Drivers and Executors

- At high level, a Spark application runs as a set of ***independent processes*** on the top of the dataset distributed across the machines of the cluster and consists of one driver program and several executors.
- The ***driver program***, hosted in the master machine, ***runs*** the user's ***main function*** and ***distributes*** operations on the cluster by sending several units of work, called tasks, to the executors.
- ***Each executor***, hosted in a ***slave machine***, runs tasks in ***parallel*** and keeps data in memory or disk storage across them.

Scheme of a typical architecture of a spark program



The **SparkContext** includes the main functions of the user and it is in charge of sending the different task to different nodes (each node may have several computing units so each node can execute different tasks).

Each node has a **cache** memory for storing in memory the dataset.

The **Cluster Manager** is in charge of managing the communication between the nodes of the clusters.

Spark RDD:

- The main abstraction provided by Spark is the ***resilient distributed dataset*** (RDD)
- RDD is a ***fault-tolerant*** collection of elements partitioned across the machines of the cluster that can be processed in parallel
- These collections are ***resilient***, because they ***can be rebuilt*** if a portion of the dataset is lost
- The applications developed using the Spark framework are ***totally independent*** of the ***file system*** or the ***database management*** system used for storing data
- Indeed, there exist ***connectors*** for reading data, creating the RDD and writing back results on files or on databases
- In the last years, ***Data Frames*** and ***Datasets*** have been recently released as an abstraction on top of the RDD.

DATABASE REVOLUTION

Timeline of Database Framework:

- *1950 – 1972 -> Pre-Relational*
- *1972 – 2005 -> Relational (Oracle, SQL Server, MySQL)*
- *2005 – 2015 -> The Next Generation (MapReduce, Hadoop, Cassandra, MongoDB)*

History

- **Pre Historic**: Database is an organized collection of data, thus *dictionaries, encyclopedias and libraries (indexed archives)* can be considered the first databases
- **Tape Revolution** (after WWII): Data were stored **sequentially** adopting *Paper* or *Magnetic Tape*.
  - *Features*: "fast forward" and "rewind" through the datasets
  - *Problems*: o direct high-speed acces to individual records
- **First Electronic Databases** (60s): first **OLTP** (On-line Transaction Processing) computer systems appeared.
  - *Main features*: **ISAM** (Index Sequential Access Method)
  - *Drawbacks*: there were databases but **no Database Management Systems (DBMS)**. The application had the **control** of the database

Working without DBMS, every application had to **reinvent the database wheel**.

Problems:

- ***Errors*** in application data handling ***code*** led inevitably to ***corrupted data***.
- Allowing ***multiple users*** to concurrently access or change data requires ***sophisticated coding***.
- ***Optimization*** of data access requires ***complicated*** and ***specialized*** algorithms that could not easily be ***duplicated*** in each application.

First Database Revolution

DBMS represented a **new layer** that allowed the **separation** between the database handling logic and the application layer.
As a result, **programmer overhead** was minimized and the **performance** and **integrity** of data access routines were ensured.

Main features:
the definition of both a **scheme** for the data to store in the database and an **access path** for navigating among data records (**Navigational Models** as *Hierarchical* and *Network Model*).

Main Issues of the Naviational Models:

- They ***ran*** exclusively on the ***mainframe computer*** systems of the day (largely IBM mainframes)
- Only ***queries*** that could be ***anticipated*** during the initial design phase were possible
- It was ***extremely difficult*** to add new data elements to an existing system
- ***CRUD*** (Create, Read, Update, Delete) operations oriented-> complex analytic queries required ***hard coding***

Second Database Revolution

Backgrounds:

- The business demands for ***analytic-style reports*** were growing rapidly
- Existing databases were ***too hard*** to use
- Existing databases ***lacked*** a ***theoretical*** foundation
- Existing databases ***mixed*** logical and physical implementations

Relational Theory Main Features:

- **Tuples**: an unordered set of attribute values.
- **Relations**: collections of distinct tuples
- **Constraints**: enforce **consistency** of the database. **Key constraints** are used to identify tuples and relationships between tuples.
- **Operations on relations** (joins, projections, unions): these operations always return relations.

The **third normal form** indicates that *"non-key attributes must be dependent on the key, the whole key and nothing but the key".*

“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 are needed to ensure **consistency** and **integrity** of data:
- ***Atomic***: The transaction is ***indivisible***—either all the statements in the transaction are applied to the database or none are.
- ***Consistent***: The database remains in a consistent state ***before*** and ***after*** transaction execution.
- ***Isolated***: While multiple transactions can be executed by one or more users simultaneously, one transaction should ***not see the effects*** of other inprogress transactions.
- ***Durable***: Once a transaction is saved to the database, its changes are expected to ***persist*** even if there is a ***failure*** of operating system or hardware.

With **Relational DBMS** and **SQL** language, the database users appreciated the possibility of **quickly** writing reports and analytics queries.

Client-Server:
- In the client-server model, ***presentation*** logic was hosted on a PC terminal typically running an operating systems with a GUI.
- Database system were hosted on, usually, ***remote servers***.
- ***Application logic*** was often concentrated on the client side, but could also be located within the database server using the ***stored procedures (***programs that ran inside the database).
- Application based on the Client-Server paradigm exploited relational DBMSs as ***backend*** and assumed SQL as the ***vehicle for all requests*** between client and server.

Object-oriented Programming VS RDBMS:
- Object-oriented (OO) developers were ***frustrated*** by the mismatch between the object-oriented representations of their data within their programs and the relational representation within the database.
- In an OO program, all the details relevant to a ***logical unit of work*** would be stored within the ***one class*** or directly linked to that class.
- When an object was stored into or retrieved from a *relational database*, ***multiple SQL operations*** would be required to convert from the object-oriented representation to the relational representation.

“A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers”

Solutions for Frustrations:

- A new DBMS model, namely ***Object Oriented Database Management System* (OODBMS)**
- OODBMS would allow to ***store*** program objects ***without normalization***
- Applications would be able to ***load and store*** their object very ***easily***
- The ***implementation*** resembles the "***Navigation***" models
- However, OODBMS systems had ***completely failed*** to gain market share
- OO programmers became ***resigned*** to the use of RDBMS systems to persist objects
- ***Object-Relational Mapping*** (*ORM*) frameworks (Hibernate for example) alleviated the pain of OO programmers

Final considerations:

- For a period of roughly ***10 years*** (1995–2005), no significant new databases were introduced
- Although in this period the ***Internet*** has started to ***pervade*** the life of each of us, no new database architectures have emerged
- However, after 2005, the ***era of massive web-scale applications*** created pressures on the relational database model: its supremacy was just about to end!

Massive web-scale applications (**MWSAs**) such as Google need:

- **large** volumes of read and write operations
- **low** latency responses time
- **high** availability

Limits of RDBMSs for MWSAs:

- The performance of RDMSs may be improved by ***upgrading*** CPUs, ***adding*** memory and ***faster*** storage devices (***vertical scaling*** of the servers) .
- Vertical scaling is ***costly***, also in terms of maintenance.
- There are ***limitations*** on the number of CPUs and memory dimension supported by a server.

Third Database Revolution

Motivations: MWSAs serving **huge amount of users** needs large-scalable DBMSs characterized by **Scalability, Flexibility, Availability** and **Low Cost**.

Scalability (Example: a spike of traffic to a website. Solution: adding a new server and then shout it down when the traffic become normal) -> *Scaling up vs Scaling out*

Flexibility (*Ex*: e-commerce application which handles several kind of items. Items of the same type may also have different descriptions and missing values.

Sol: adding/removing/modifying one or more fields which describe the items in the database) -> *Schema vs Schema-less models*

Availability (*Ex*: websites such as social networks or e-commerce services must be always available for users. Frequently out-of-services are not permitted for their business. *Sol*: to deploy the whole systems on multiple servers. If one server goes down, the services continue to be available for users, even though performances (and also data consistency) may be reduced) -> *Replication and Sharding*

Costs (*Ex*: most of RDBMSs used by enterprises and organizations have often licensing costs. These costs depends on the number of users. *Sol*: to use open source DBMSs. Most of the systems born during the third revolution offers open source solutions. Often, companies (both the same offering the solution or third party) offer consulting services for technical supports)

Most of NoSQL solutions **ensures** Scalability, Availability and Flexibility and are often **open source**.

In **NoSQL** databases ACID transactions may be **not supported**.

Roughly speaking, NoSQL databases **reject** the constraints of the **relational model**, including the strict consistency and schemas.

Databases have to store data **Persistently**, mantain data **Consistency**, ensure data **Availability**. The **DBMSs** is in charge of these three tasks.

Most of the recent NoSQL DBMSs can be deployed and used on **distributed systems**,namely on multiple servers rather than a single machine.

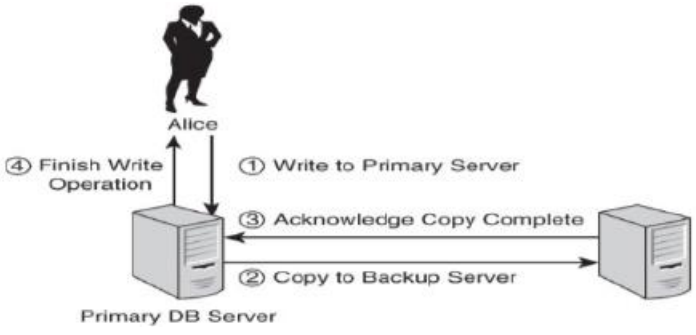
Pros of Distributed Systems:

- Distributed systems ensure **scalability, flexibility, cost control** and **availability**.
- It's heasier to add or remove nodes (**horizontal scalability**) rather than to add memory or upgrade the CPUs of a single server (**vertical scalability**).

- Allow the implementation of **fault tolerance** strategies.
- Accomplish with the motivations that led to the **third databases revolution**

Distributed systems **soffer** in **balancing** the requirements of data **consistency** and system **availability** and they **have to** protect themselves from **network failures** that may leave some nodes **isolated**:

- Data in the distributed servers must be stored in a way that is **not lost** when the database server is shut down (**Data Persistency**).
- Data stored in a single server DBMS may be not available for several reasons (failures of the OS, voltage drops, disks break down). A possible solution to **ensure** high data availability is the **two-phase commit**

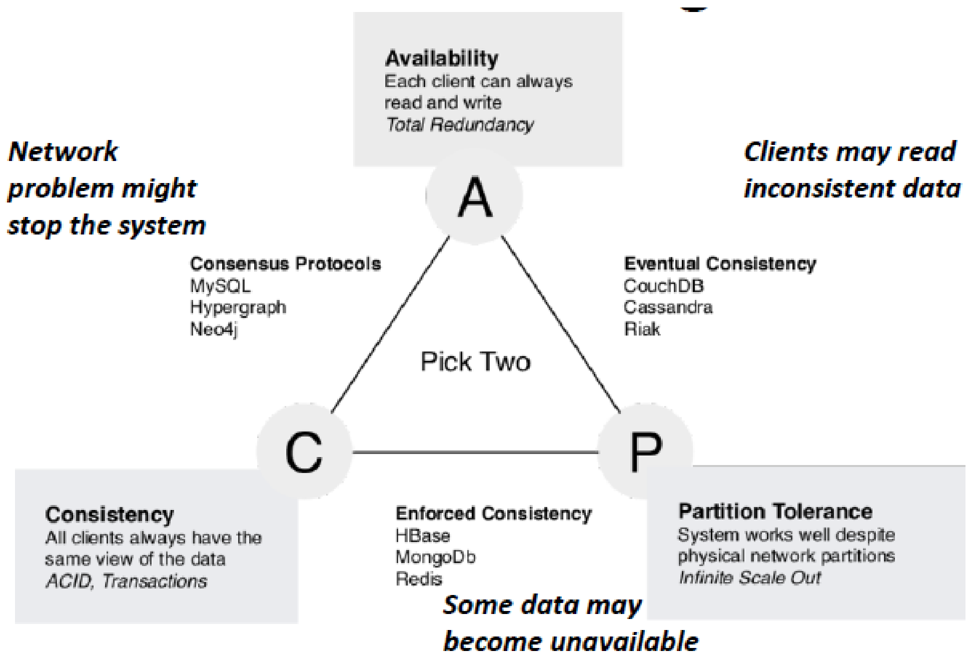
  If the primary server **goes down**, the **Backup Server** takes its place.
  The problem with this solution is that the two-phase commit **is a transaction**, so it takes longer time to be completed respect to writing on a single server if we want to have both **consistent** and **highly available data**.
  The two-phase commits **favors consistency over availability**; while the two-phase commit is executing, other queries to the data are **blocked**, so the updated data is unavailable until the commit finishes

In some applications, **ACID** transactions lead us to have not acceptable **waiting times** (we need a fast response).

Network Partition: the system has ***two choices***: either show each user a different view of the data (availability but not consistency) , or shut down one of the partitions and disconnect one of the users (consistency but not availability).

The **CAP Theorem** says that distributed databases cannot ensure at the same time **Consistency(C), Availability(A)** and **Partition Protection(P)** (i.e. if a network that connects two or more databases servers fails, the servers will still be available).

At maximum **two** of the previous features may be found in a distributed database.


Availability
Each client can always read and write
Total Redundancy
Network problem might stop the system
Clients may read inconsistent data
A
Consensus Protocols
MySQL
Hypergraph
Neo4j
Eventual Consistency
CouchDB
Cassandra
Riak
Pick Two
C
P
Consistency
All clients always have the same view of the data
ACID, Transactions
Enforced Consistency
HBase
MongoDb
Redis
Partition Tolerance
System works well despite physical network partitions
Infinite Scale Out
Some data may become unavailable

Solutions:

- **CA**: **site cluster**, therefore all nodes are always in contact, when a partition occurs, the system blocks. Total consistency can affect performance (latency) and scalability.
- **AP**: system is still available under **partitioning**, but some of the data returned may be inaccurate.
  This solution may return the **most recent version** of the data you have, which could be **stale**. This system state will also **accept writes** that can be processed later when the partition is resolved.
  Availability i also a compelling option when the system needs to continue to function in spite of **external errors**.
- **CP**: some data may not be accessible, but the resti s still consistent/accurate.

BASE properties of NoSQL databases:

- **BA** stands for *Basically Available*. Partial failures of the distributed database may be handled in order to ensure the availability of the service (often thanks to replication)
- **S** stands for *Soft state*. Data stored in the nodes may be updated with more recent data because the eventual consistency model (no user writes may be responsible of the updating)
- **E** stands for *Eventually consistent*. At some point in the future, data in all nodes will converge to a consistent state

High availability is a strong requirement of modern shared-data systems.

To achieve it, data and services must be **replicated**.

Replication impose **consistency maintenance**.

Every form of consistency requires communication and a stronger consistency required **higher latency**.

Types of Eventual Consistency

• ***Read-Your-Writes Consistency*** ensures that once a user has updated a record, all of his/her reads of that record will return the updated value.
• ***Session consistency*** ensures "read-your-writes consistency" during a session. If the user ends a session and starts another session with the same DBMS, there is no guarantee the server will "remember" the writes made by the user in the previous session.
• ***Monotonic read consistency*** ensures that if a user issues a query and sees a result, all the users will never see an earlier version of the value.
• ***Monotonic write consistency*** ensures that if a user makes several update commands, they will be executed in the order he/she issued them.
• If an operation ***logically depends*** on a preceding operation, there is a causal relationship between the operations. ***Causal consistency*** ensures that clients will observe results that are consistent with the causal relationships.

KEY VALUE DATABASES (*KVD*)

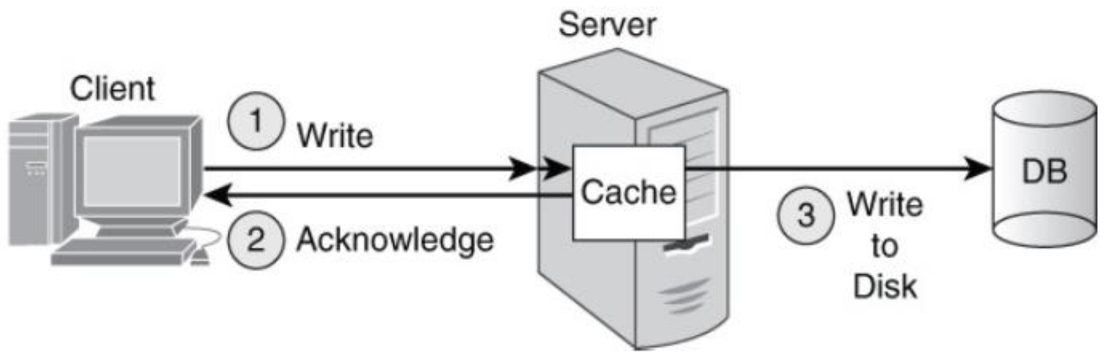
There is **no need** for **tables** if we do not need groups of related attributes.

The unique requirement is that **each value** has a **unique identifier** in the form of the key.

Keys must be **unique** within the **namespace** defined in each specific **bucket**.

Essential features of *KVD*:

- **Simplicity**: do **not** have to **define** a database **schema** nor define data **types** for each **attribute**.
  KVD are **flexible** and **forgiving**: we can make mistakes assigning wrong type of data to an attribute or use different type of data for a specific attribute.
- **Speed**: using the schema below, the *KVD* can **write** the recently updated value to disk, while the user program is doing something else

  Read operations can be **faster** if data is stored **in memory**; this is the motivation for using **cache**. When the *KVD* **uses all the RAM memory** allocated to it, the database will need to **free** some records; a typical algorithm for selecting the records to remove is the **LRU** (*Least Recently Used*)
- **Scalability**: in the framework of *KVD*, two main approaches are adopted to handle clusters:
  - **Master-Slave Replication**: the master accepts **writes** and **reads** and is in charge of **collecting** all the **modification** of the data and **updating** data to all the slaves. The **slaves** may only respond to **read requests**.
    <<<This architecture is specially **suitable** for applications with a **growing demand for read operations**>>>
    - **Pros**: very **simple** and **no need to coordinate** writes or **resolve** conflicts between multiple servers accepting writes

- **Cons**:
    - It suffers increasing number of writes (only the master accepts writes and all the updates must be sent to the slaves).
    - If mster fails, the cluster cannot accept writes: a solution to this problem could be to implement a protocol for the slaves to **promote** a **new master** in the case the old master **fails** (doesn't respond to messages within some period of time)

Masterless Replication: All nodes may accept write requests<<<**suitable** for applications in which the number of **writes** may increase and achieve peaks>>>
It can exploit a **ring structure abstraction**.
Each **node** is in **charge** of helping its **neighbors.**

- **Handling Replication**: Each server replicates to its neighbors

Developers tends to use key-value databases when **ease of storage** and **retrieval** are more important than organizing data into more complex data structures, such as tables or networks.

A **Namespace** is a collection of key-value pairs that has no duplicate keys. It is allowed to have **duplicate values** in a namespace.

They are helpful when **multiple applications** use the same *KVD*.

Namespaces allows to organize data into **subunits**.

We may define **interpretable** keys using a structure like **EntityName:EntityID:EntityAttribute**.

Keys may be strings, numbers, list or other complex structures. We use **Hash functions** to obtain the address of the value from an **arbitrary key**.

Strings representing keys should **not** be **too long**. Long keys will use more memory and *KVD* tend to be **memory-intensive** systems.

At the same time, avoid keys that are **too short**. Short keys are more likely to lead to **conflicts** in key names.

<u>Hashing for Selecting Servers</u>

We would like to **balance** the write loads among the servers in a masterless replication architecture.

If we have ***N*** servers, we can give each of them a **unique ID**.

We can then take a **key**, apply to it the **hashing function**, **divide** the hash value by ***N***, collect the reminder (**modulus**) and use the result so assign the **value** represented by the initial key to the server with the same **ID**.

In *KVD* we can have operations to **retrieve, set** and **delete**. For more complex queries, we have to do them with an application program.

Limitations

- ***The only way to look up values is by key.***
- ***Some key-value databases do not support range queries.***
- ***There is no standard query language comparable to SQL for relational Databases.***

If ***data organization*** and ***management*** is more important than the performances, classical relational databases are more suitable rather than key-value databases. However, if we are more interested to the ***performances*** (high availability, short response time, etc.) and/or the data model is not ***too much complicated*** (no hierarchical organization, limited number of relationships) we may use key-values databases. Indeed, key-value stores are really ***simple*** and ***easy*** to handle, data can be modeled in a less complicated manner than in RDBMS.

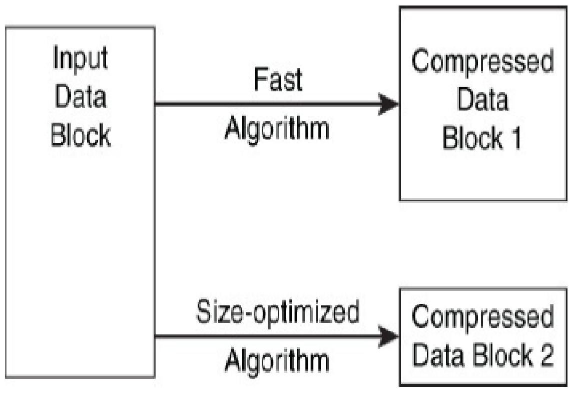
Data Compression for KV Databases

Key-value databases are ***memory intensive.***

Operating systems can exploit ***virtual memory*** management, but that entails writing data to disk or flash storage.

Reading from and writing to ***disk*** is significantly ***slower*** than reading from RAM memory.

One way to optimize memory and persistent storage is to use data ***compression techniques***.



Look for compression algorithms that ensure a ***trade-off*** between the ***speed*** of compression/decompression and the ***size*** of the compressed data.

DATA PARTITIONING

Subsets of data (**partitions** or **shards**) may be handled by different nodes of a cluster.

A cluster may contain **more than one** partition

The **main objective** of partitioning data would be to **evenly balance** both writes and reads loads among servers.

If needed, **additional** nodes would be **easily** added to the cluster and data appropriately **relocated**.

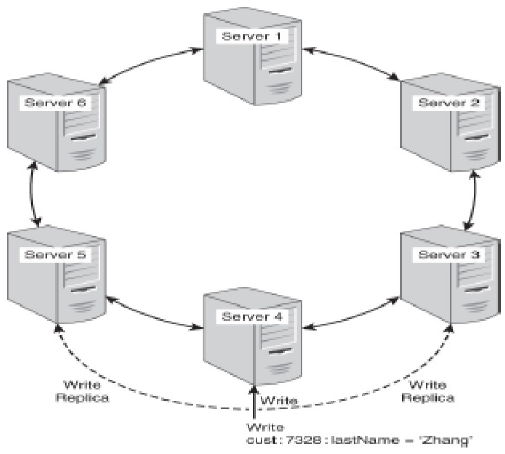
A **partition key** identifies the specific partition in which the value has been stored.

Usually, hashing functions are adopted for actually **identifying** the specific cluster or partition.

Clusters

- In *KVD*, clusters tend to be **loosely coupled**; servers **are fairly independent** and complete many functions on their own with **minimal coordination** with other servers in the cluster
- Each server is **responsible** for the operations on **its own partitions** and routinely **send messages** to each other to indicate they are still functioning
- When a node **fails**, the other nodes in the cluster can respond by **taking over the work** of that node

Ring Logical Structure for Organizing Partitions



Use modulo operation to decide which server is in charge of a specific value.

Whenever a value is written to a server, it is also written to the two servers linked to it (**high availability**).

If a server **fails**, the ones linked to it could **respond** to read/write requests in its stead; when it comes **back online**, the linked servers **can update it**.

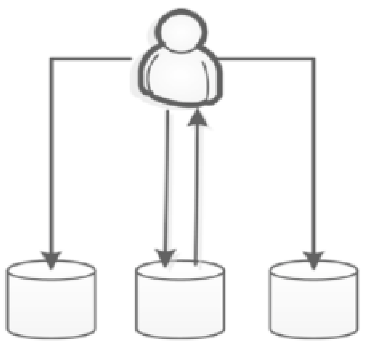
REPLICATION

High Availability is ensured by using **replication** (saving **multiple copies** of the data in the nodes of a cluster).

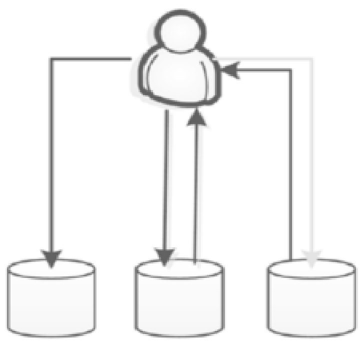
The **higher** number of **replicas**, the **less** likely we will **loose data**, the **lower** the **perofmance** of the systems (in terms of response time) **and viceversa**.

A **low** number of replicas may be used whenever **data** is easily **regenerated** and **reloaded**.

Write/Read Operations with Replicas



N=3 W=3 R=1
Slow writes, fast reads, consistent
There will be 3 copies of the data.
A write request only returns when all 3 have written to disk.
A read request only needs to read one version.

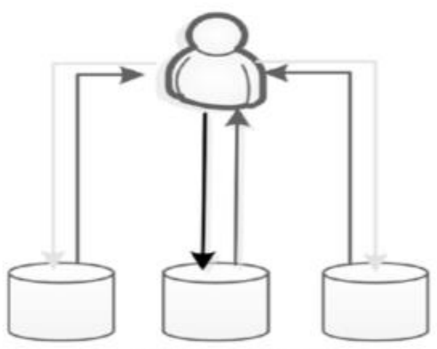


N=3 W=2 R=2
Faster writes, still consistent (quorum assembly)
There will be 3 copies of the data.
A write request returns when 2 copies are written – the other can happen later.
A read request reads 2 copies make sure it has the latest version.

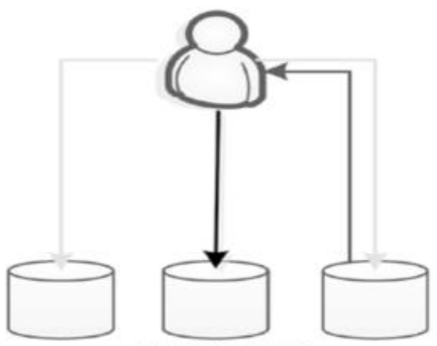
N = # of replicas

W = # of copies to be written before the write can complete

R = # of copies to be read for reading a data record



N=3 W=1 R=N
Fastest write, slow but consistent reads
There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads all copies to make sure it gets the latest version.
Data might be lost if a node fails before the second write.



N=3 W=1 R=1
Fast, but not consistent
There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads a single version only: it might not get the latest copy.
Data might be lost if a node fails before the second write.

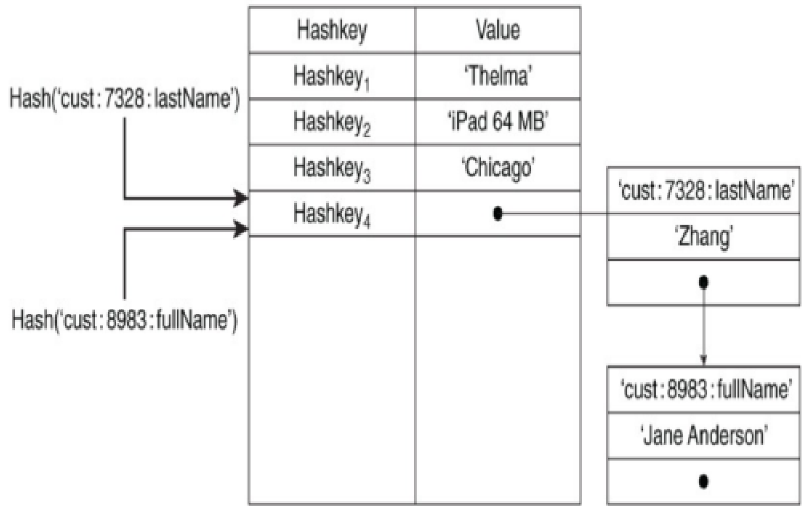
HASH FUNCTIONS

Even small changes in the input can lead to **large changes** in the output.

Generally designed to **distribute** inputs **evenly** over the set of all possible outputs.

So, **no matter how similar** two keys are, they are evenly distributed across the range of possible output values.

Collision Resolution Strategy: from a logical point of view, the table that projects a hashed key to the corresponding value may include a list of values. In each block of the list, also the original key must be present.

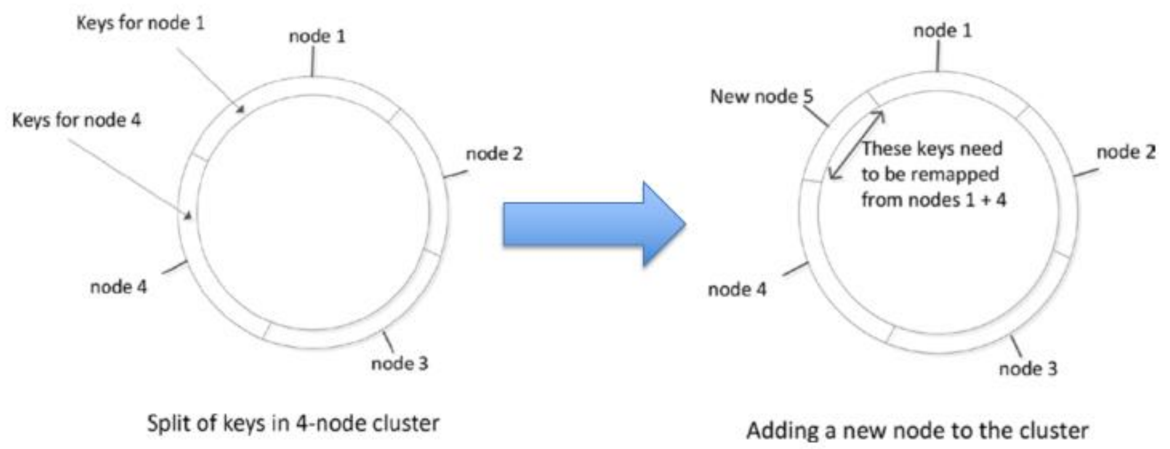


Consistent Hashing

Use Case: adding or removing a node

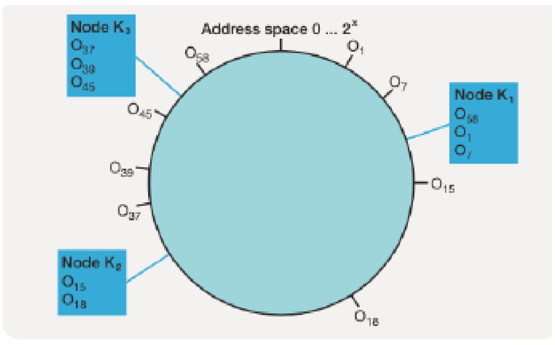
Problem: we need to change the **hashing function** and to re-locate all the data among the servers of the cluster

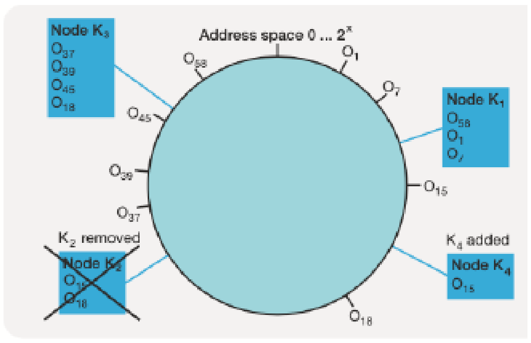
Solution: to exploit the ring structure and a **consistent hashing function** that allows us to remap only the keys mapped to the neighbors of the new node


Keys for node 1
node 1
Keys for node 4
node 2
node 4
node 3
Split of keys in 4-node cluster
node 1
New node 5
These keys need to be remapped from nodes 1 + 4
node 2
node 4
node 3
Adding a new node to the cluster

Consistent hashing ensures a good load balance among servers

The same hashing function is applied both to the **keys** and the **server/partition ID**; the hash values must be in the **same range**





The actual server (Node) *Kj* associated to a specific key (Object) *Oi* is its **successor** in the hashing/address space.

DOCUMENT DATABASES (*DD*)

Main Features

- **Non-relational** databases that store data as structured documents (**XML, JSON** …)
- They ensure **High Flexibility** (**schema-less**) level and allow also to handle **complex** data **structures**
- They allow **complex operations**, such as queries and filtering
- Some of them allow **ACID transactions**

XML vs JSON

- **XML:**
  - **Advantages**:
    - Text (Unicode) bases; can be transmitted efficiently
    - One XML document can be displayed differently in different media and software platforms
    - XML documents can be modularized; parts can be reused

- **Drawbacks**:
  - XML tags are **verbose** and **repetitious**, thus the amount of storage required increases
  - XML documents are **wasteful of space** and **computationally expensive** to parse
  - In general, XML databases are used as **content-management systems**: collection of text files are organized and mantained in XML format
  - **JSON-based** *DD* are more suitable to support **web-based** operational **workloads**, such as storing and modifying dynamic contents

- **Comparison of JSON and XML:**
  - **Similarities**:
    - Both are human readable
    - Both have very simple syntax
    - Both are hierarchical
    - Both are language independent
    - Both are supported in APIs of many programming languages
  - **Differences**:
    - Syntax is different
    - JSON is less verbose
    - JSON includes arrays
    - Names in JSON must not be JavaScript reserved words

JSON Databases

A **documenti s** the basic **unit** of storage; it can contain **nested documents** and **arrays**, which may also contain **documents** allowing for a complex hierarchical structure.

A collection, or **data bucket**, is a set of documents sharing some **common purpose**.

The document in a collection may be different (**Polymorphic Scheme**).

JSON Databases are **schema-less**: predefined document elements must not be defined.

A schema is a specification that describes the **structure of an object**.

Schema-less

- **Pros**: High flexibility in handling the structure of the objects to store
- **Cons:** the DBMS may be not allowed to enforce rules based on the structure of the data (i.e. check if all required fields are present, the values are in expected ranges ecc.).

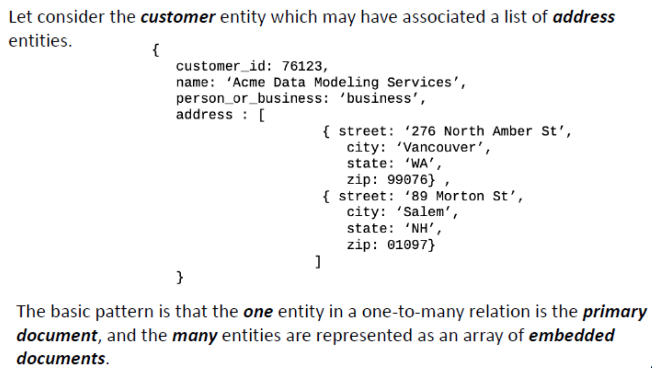
Nested documents are used for representing **relationships** among the different entities.

Document database **do not** generally provide **join operations**.

Some JSON databases have some **limitations** of the maximum **dimension** of a single document.

When modeling data for *DD*, there is no equivalent of third form that defines a "correct" model; the **nature of the queries** to be executed **drives** the approach to **model** data.

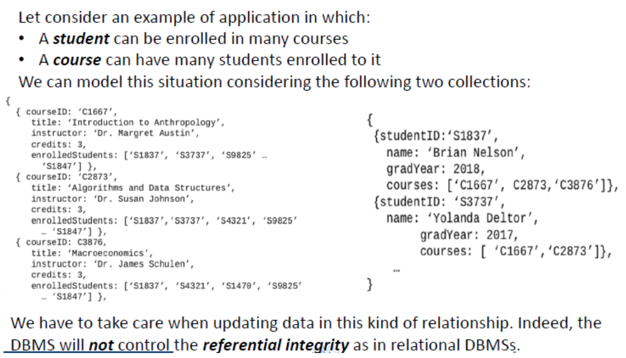
Data Modeling



Let consider the ***customer*** entity which may have associated a list of ***address*** entities.

```
{
    customer_id: 76123,
    name: 'Acme Data Modeling Services',
    person_or_business: 'business',
    address : [
                  { street: '276 North Amber St',
                    city: 'Vancouver',
                    state: 'WA',
                    zip: 99076} ,
                  { street: '89 Morton St',
                    city: 'Salem',
                    state: 'NH',
                    zip: 01097}
                 ]
}
```

The basic pattern is that the ***one*** entity in a one-to-many relation is the ***primary document***, and the ***many*** entities are represented as an array of ***embedded documents***.



Let consider an example of application in which:

- A ***student*** can be enrolled in many courses
- A ***course*** can have many students enrolled to it

We can model this situation considering the following two collections:

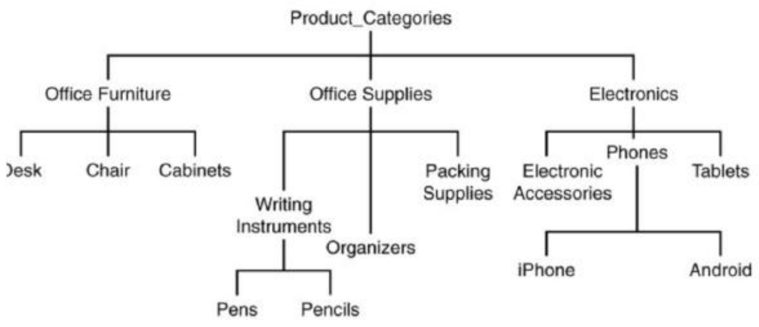
```
{
 { courseID: 'C1667',
   title: 'Introduction to Anthropology',
   instructor: 'Dr. Margret Austin',
   credits: 3,
   enrolledStudents: ['S1837', 'S3737', 'S9825' …
     'S1847'] },
 { courseID: 'C2873',
   title: 'Algorithms and Data Structures',
   instructor: 'Dr. Susan Johnson',
   credits: 3,
   enrolledStudents: ['S1837','S3737', 'S4321', 'S9825'
     … 'S1847'] },
 { courseID: C3876,
   title: 'Macroeconomics',
   instructor: 'Dr. James Schulen',
   credits: 3,
   enrolledStudents: ['S1837', 'S4321', 'S1470', 'S9825'
     … 'S1847'] },
```

```
{
 {studentID:'S1837',
   name: 'Brian Nelson',
   gradYear: 2018,
   courses: ['C1667', C2873,'C3876']},
 {studentID: 'S3737',
   name: 'Yolanda Deltor',
        gradYear: 2017,
        courses: [ 'C1667','C2873']},
     …
}
```

We have to take care when updating data in this kind of relationship. Indeed, the DBMS will ***not*** control the ***referential integrity*** as in relational DBMSs.

One to Many *Many to Many*

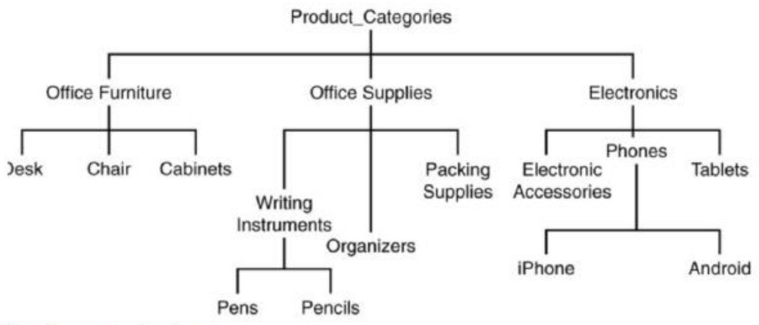
Modeling Hierarchies



Parent reference solution:

```
{
   {productCategoryID: 'PC233', name:'Pencils',
     parentID:'PC72'},
   {productCategoryID: 'PC72', name:'Writing Instruments',
     parentID: 'PC37"},
   {productCategoryID: 'PC37', name:'Office Supplies',
     parentID: 'P01'},
   {productCategoryID: 'P01', name:'Product Categories' }
}
```

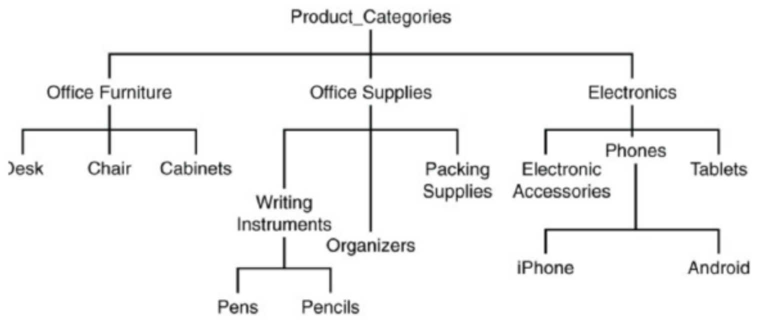
This solution is useful if we have frequently to show a specific instance of an object and then show the more general type of that category.



Child reference solution:

```
{
  {productCategoryID: 'P01', name:'Product Categories',
       childrenIDs: ['P37','P39','P41']},
      {productCategoryID: 'PC37', name:'Office Supplies',
       childrenIDs: ['PC72','PC73','PC74"]},
       {productCategoryID: 'PC72', name:'Writing
         Instruments', childrenIDs: ['PC233','PC234']'},
        {productCategoryID: 'PC233', name:'Pencils'}
   }
```

This solution is useful if we have frequently to retrieve the children (or sub parts) of a specific instance of an object.



List of ancestor solutions:

```
{productCategoryID: 'PC233', name:'Pencils',
  ancestors:['PC72', 'PC37', 'P01']}
```

This solution allows to retrieve the full path of the ancestors with one read operation.

A change to the hierarchy may require many write operations, depending on the level at which the change occurred.

Planning for Mutable Documents: when a document is created, the DBMS allocates a certain amount of spaces for it. If the document **grows more** than the allocated space, the DBMS has to **relocate** it and **free** the previously allocated space; these steps can adversely affect the system **performance**.

A solution for avoiding to move oversized documents is to **allocate sufficient space** at the moment in which the documents is **created**.

Store different types of documents in the same collection **only if** they will be **frequently used together** in the application (they will be stored in the same **data block** in the **disk**, which can lead to **inefficiencies** whenever data is read from disk but not used by the application).

The **structure** of the documents should depend on the **queries** we intend to do on the database.

Normalization vs Denormalization

- **Normalization** helps **avoid** data **anomalies**, but it can cause **performance problems**
- With **normalized** data we need **join operations**, which must be optimized for improving performances
- If we use **denormalized** data, we may introduce **redundancies** and cause anomalies
- On the other hand, we may **improve the performances** of the queries because we **reduce** the number of collectins and **avoid join operations**
- **Denormalization supports** improving read operations when **indexes** are adopted

Indexing Document Database

In order to avoid the entire scan of the overall database, DBMSs for document databases (i.e. MongoDB) allow the definition of **indexes**.

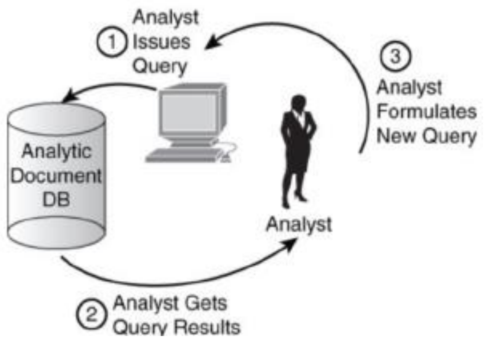
Indexes are **structured set of information** that maps from one attribute to related information.

In general, indexes are **special data structures** that store a small portion of the collection's data set in an **easy to traverse** form.

The index stores the value of a specific field or set of fields, **ordered by the value of the field**.

The ordering of the index entries supports **efficient equality matches** and **range-based query operations**.

Read-Heavy applications



*The classical scheme of read-heavy application (**business intelligence** and **analytics applications**)*

In this kind of applications, the use of **several indexes** allows to quickly access to the database; i.e indexes can be defined for easily retrieve documents describing objects related to a specific **geographical region** or to a specific **type**.

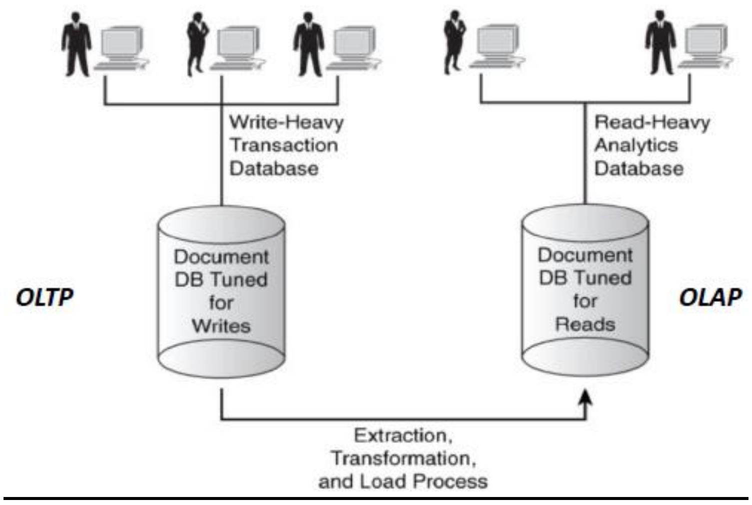
Write-Heavy applications

The **higher** the number of **indexes** adopted, the **higher** the amount of **time** required for closing a **write** operation (**all** the indexes must be **updated** and **created** at the beginning).

Reducing the number of indexes allow us to obtain systems with **fast write** operations responses. On the other hand, we have to accept to deal with **slow read operations**.

In conclusion, the number and type of indexes to adopt must be identified as a **trade-off** solution.

Transactions Processing Systems


Write-Heavy Transaction Database
Read-Heavy Analytics Database
OLTP
Document DB Tuned for Writes
Document DB Tuned for Reads
OLAP
Extraction, Transformation, and Load Process

Systems designed for **fast write** operations and **trargeted reads**, as shown in the figure.

SHARDING

Sharding, or **horizontal partitioning**, is the process of dividing data into blocks or **chuncks**.

Each block (**shard**) is deployed on a **specific node** (server) of a **cluster**.

Each node can contain **only one** shard.

In case of **data replication**, a shard can be hosted by more than one node.

Advantages

- Allows handling **heavy loads** and the **increase** of system users
- Data may be **easily distributed** on a variable number of servers that may be **added** or **removed** by request
- Cheaper than **vertical scaling**

- Combined with **replications**, ensures a **high availability** of the system and **fast response**

To implement sharding, we have to select a **shard key** and a **paritioning method**.

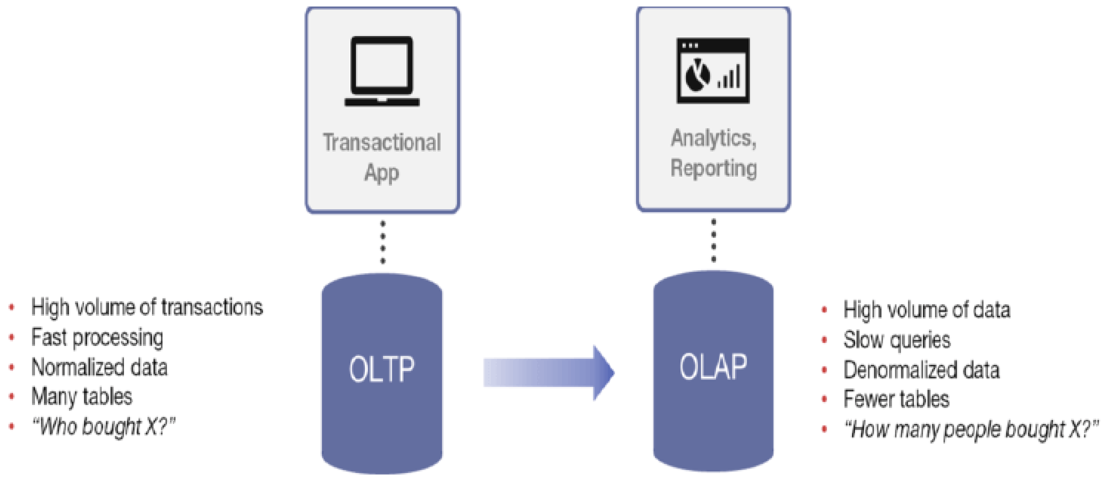
A shard key is **one or more** fields, that exist **in all documents** in a collection that is used to separate documents.

Partition Algorithms

- ***Range***: for example, if all documents in a collection had a creation date field, it could be used to partition documents into monthly shards.
- ***Hashing***: a hash function can be used to determine where to place a document. ***Consistent hashing*** may be also used.
- ***List***: for example, let imagine a product database with several types (electronics, appliances, household goods, books, and clothes). These product types could be used as a shard key to allocate documents across five different servers.

INTRODUCTION TO COLUMN DATABASES

- **OLTP** (*Online Transaction Processing*): software architectures oriented towards **handling ACID transactions**
- **OLAP** (*Online Analytics Processing*): software architectures oriented towards **interactive** and **fast analysis** of data. Typical of **Business Intelligence** software



OLTP processing is mainly oriented towards handling **one record** at a time processing.

When the **first relational databases** were designed, the world was mainly experiencing the OLTP era.

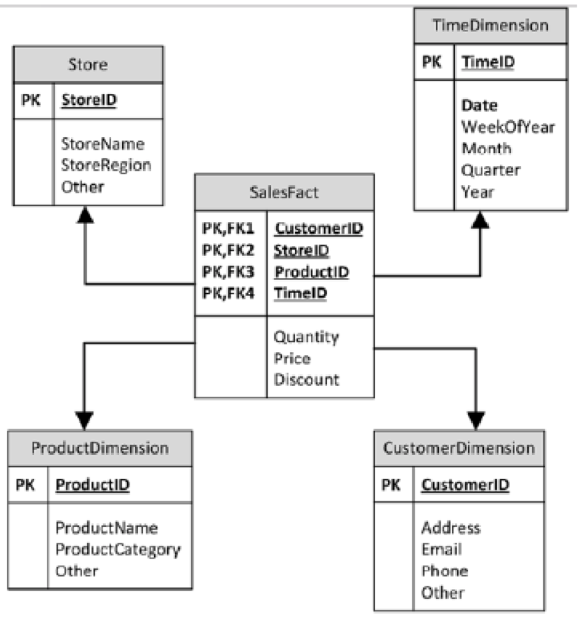
In the **record-based processing era** (up to the 80s), **CRUD** operations were the most time-critical ones. The **time response** to the queries was **not** a critical **issue**.

When **Business Intelligence** software started to spread, **OLAP** processing assumed a **big relevance**, thus the **time response** to queries became a **critical issue** to appropriately handle.

Data Warehouse

relational databases tasked with supporting **analytic** and **decision support** applications (80s-90s)

Star Schemas in Data Warehouse



nage extracted from:"Guy Harrison, Next

It is a ***relational solution*** where central ***large fact*** tables are associated with ***numerous smaller*** *dimension* tables.

Aggregate ***queries*** could execute ***quickly***.

Star Schemes do ***not*** represent a fully ***normalized*** relational model of data (redundancies are often accepted, information some times depend partly on the primary key).

Main Drwabacks of Star Schemas:

- **Data processing** in data warehouses remained severely **CPU** and **IO intensive**

- The **data volumes** grew up along with and **user demands** for **interactive** response times

- In general, around the 90s of the last century, the **discontent** with traditional data warehousing performance **increased**

The Columnar Storage

When processing data for making **analytics**, in most cases, we are not interested in retrieving **all the information** of each single records.

When dealing with **row-based** storage of records, we have to **access** to **all the records** of the considered set for retrieving just the values of one attribute.

If all the values of an attribute are **grouped** together on the **disk** (or in the block of a disk), the task of retrieving the values of one attribute will be **faster** than a row-based storage of records.

Columnar Compression

Data compression algorithms basically **remove redundancy** with data values; the higher the redundancy, the higher the compression ratios.

To find redundancy, those algorithms usually try to **work** on **localized** subsets of the data.

If data are stored in column databases, very **high compression ratio** can be achieved with very **low computational overheads**; as example, often in *CD* data are stored in a **sorted order** and in this case very high compression ratios can be achieved simply by representing each column values as a "*delta*" from the preceding column value

---

COLUMN DATABASES (*CD*)

CD perform **poorly** during **single-row modifications**.

The **overhead** needed for **reading** rows can be **partly reduced** by caching and multicolumn projections (storing multiple columns together on disk).

In general, handling a row means accessing to more than once blocks of the disk.

Differences with Relational DBs:

- Column databases are implemented as **sparse** and multidimensional **maps**
- Columns can **vary** between rows
- Columns can be added **dynamically**
- **Joins** are **not** used because **data** is **denormalized** for reducing the number of tables to more than one table
- It is suggested to use **separate keyspace** for each application

Delta Store

Problem: The basic column storage architecture is ***unable*** to cope with ***constant*** stream of ***row-level modifications***.

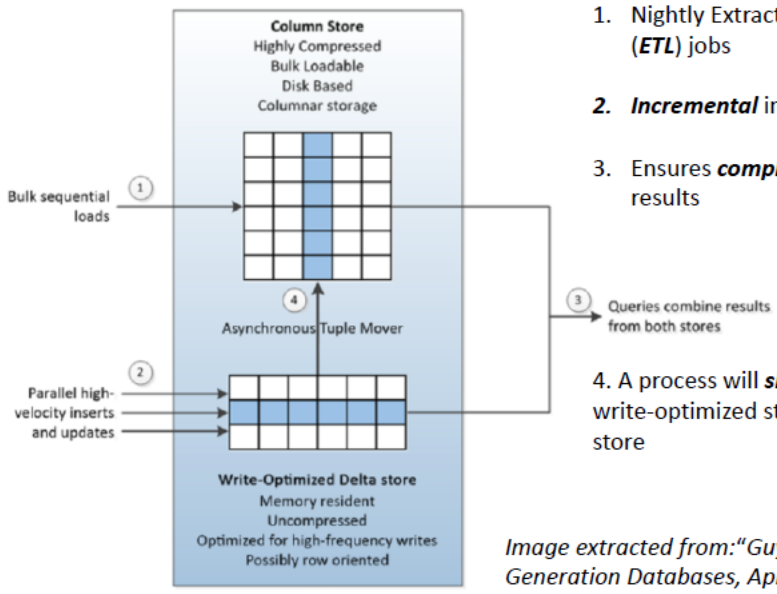
These modifications are needed whenever ***data warehouse*** have to provide real-time "***up-to-the-minute***" information.

Solution: Delta store is an area of the database, resident in ***in-memory uncompressed*** and optimized for high-frequency data modifications.

Data in the delta store is ***periodically merged*** with the main ***columnar***oriented store (often data are ***highly compressed***).

Queries may need to ***access both*** the delta store and the column store in order to return complete and accurate results.

Delta Store: Architecture



1. Nightly Extract, Transform, Load (***ETL***) jobs

2. Incremental inserts and updates

3. Ensures ***complete*** and ***consistent*** results

4. A process will ***shift data*** from the write-optimized store to the column store

Image extracted from:"Guy Harrison, Next Generation Databases, Apress, 2015"

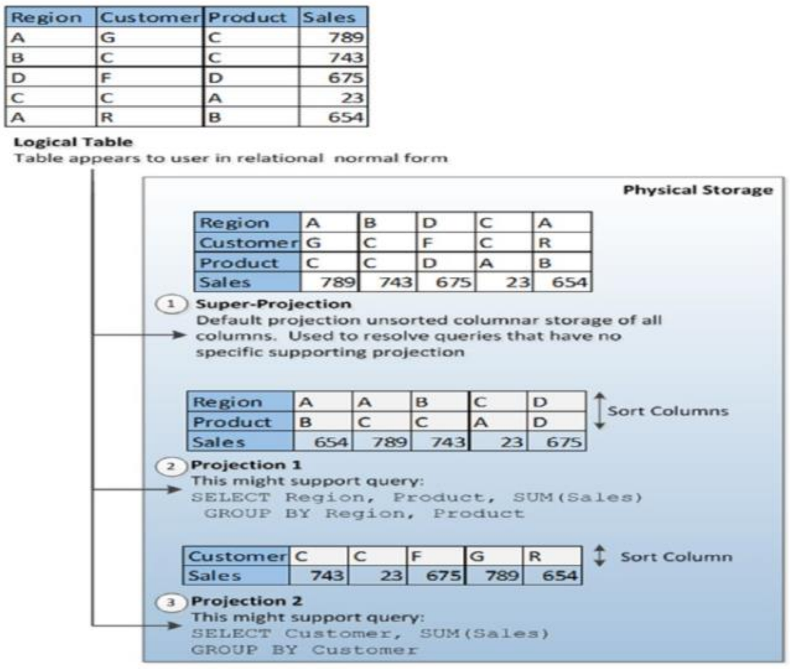
Projections

Problem: Complex queries often need to read ***combinations*** of column data.

Solution: Some column databases adopts ***projections***.

Projections: combinations of columns that are ***frequently*** accessed together.

Columns of specific projections are ***stored together*** on the disk.



Region	Customer	Product	Sales
A	G	C	789
B	C	C	743
D	F	D	675
C	C	A	23
A	R	B	654

Logical Table
Table appears to user in relational normal form

Physical Storage

Region	A	B	D	C	A
Customer	G	C	F	C	R
Product	C	C	D	A	B
Sales	789	743	675	23	654

1 **Super-Projection**
Default projection unsorted columnar storage of all columns. Used to resolve queries that have no specific supporting projection

Region	A	A	B	C	D
Product	B	C	C	A	D
Sales	654	789	743	23	675

Sort Columns

2 **Projection 1**
This might support query:
```
SELECT Region, Product, SUM(Sales)
 GROUP BY Region, Product
```

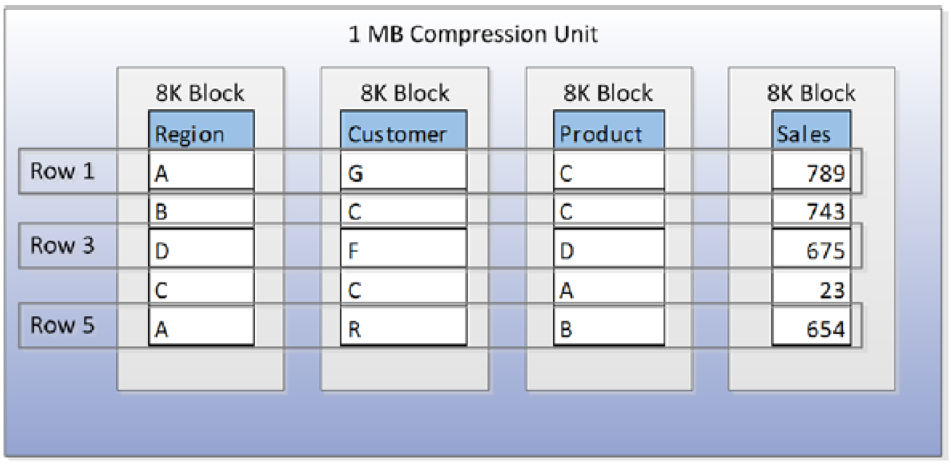
Customer	C	C	F	G	R
Sales	743	23	675	789	654

Sort Column

3 **Projection 2**
This might support query:
```
SELECT Customer, SUM(Sales)
GROUP BY Customer
```

example of database architecture based on projections

Hybrid Columnar Compression (Oracle Solution)



- Rows of data are contained within compression units of about 1 MB
- Columns stored together within smaller 8K blocks

With the NoSQL databases discusses so far we can resolve the problem of handling **huge amount** of data, but:

- ***KVD* lack support** for organizing **many columns** and keeping **frequently used** data together.
- ***DD* may not** have some of the useful features of relational databases, such as **SQL-like queries** language.

BigTable: The Google NoSQL Database

It was introduced in 2006 for Google large services such as Google Earth and Google Finance.

Main Features:

- ***Column based*** with a ***dynamic*** control over columns
- Data values are indexed by ***row*** identifier, ***column*** name, and ***a time stamp***
- Reads and writes of a row are ***atomic***
- ***Rows*** are maintained in a ***sorted order***

BigTable: An Example

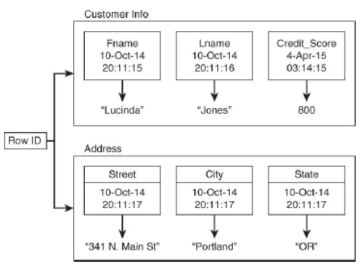


Image extracted from: "Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015

- A data record is a ***row*** composed of several ***column families***.
- Each family consists of a set of ***related columns*** (frequently used together.)
- ***Column values*** within a column family are ***kept together*** on the disk, whereas values of the same row belonging to different families may be stored far.
- ***Columns families*** must be defined ***a priori*** (like relational tables).
- Applications may ***add*** and ***remove columns*** inside families (like key-value pairs).

BigTable: Indexing

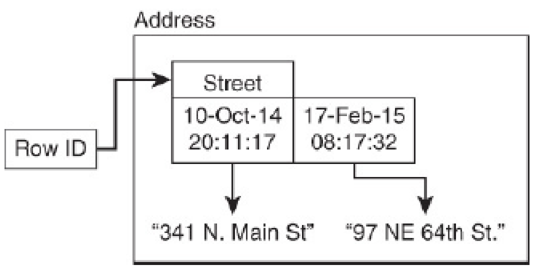


Image extracted from: “Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015”

A data values is indexed by its :

- ***row identifier*** (similar role than a primary key in relational databases)
- ***column name***, which identifies a column (similar role than a key in key-value databases)
- ***time stamp***, which allows the existence of multi versions of a column value can exist.

A **Keyspace** is a **top-level logical container**. It holds families, row keys and related data structures. Typically there is one keyspace per application.

Row keys can also be used to **partition** and **order** data.

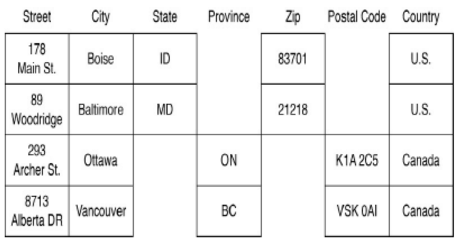
Algorithms for **data partitioning** and **ordering** depends on specific databases (i.e. **Cassandra** adopts the ***partitioner***, a specific object which is used for sorting data).

By default, the partitioner **randomly distributes** rows across nodes.

Column Families

Columns that are ***frequently*** used together are ***grouped*** into the same column family.

Column family ***analogous*** to a ***table*** in a relational database. However:



Street	City	State	Province	Zip	Postal Code	Country
178 Main St.	Boise	ID		83701		U.S.
89 Woodridge	Baltimore	MD		21218		U.S.
293 Archer St.	Ottawa		ON		K1A 2C5	Canada
8713 Alberta DR	Vancouver		BC		VSK 0AI	Canada

Image extracted from: “Dan Sullivan, NoSQL For

- Rows in columns families are usually ***ordered*** and ***versioned***
- Columns in a family can be modified ***dynamically***.
- Modifications just require making a reference to them from the ***client application***.

Cluster and Partitions

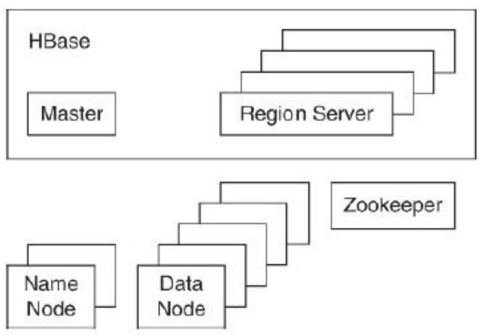
The most recent column databases are designed for ensuring high ***availability*** and ***scalability***, along with different ***consistency levels***.

Clusters and partitions are fundamental for the ***distributed implementation*** of the databases.

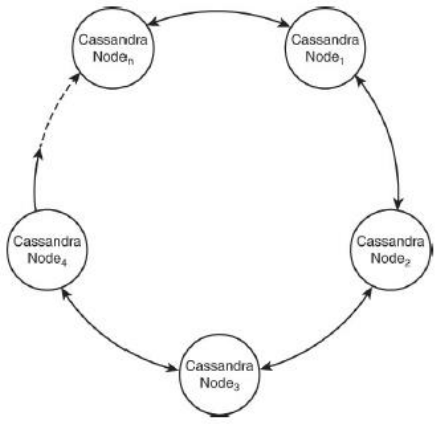
A partition is a ***logical subset*** of a database. Partitions are usually used to store a set of data based on ***some attribute*** of the data in column databases, they can be generated on the basis of:

- A ***range of values***, such as the value of a row ID
- A ***hash value***, such as the hash value of a column name
- A ***list of values***, such as the names of states or provinces
- A ***composite*** of two or more of the above options

Column DB Architectures



Apache HBase Architecture



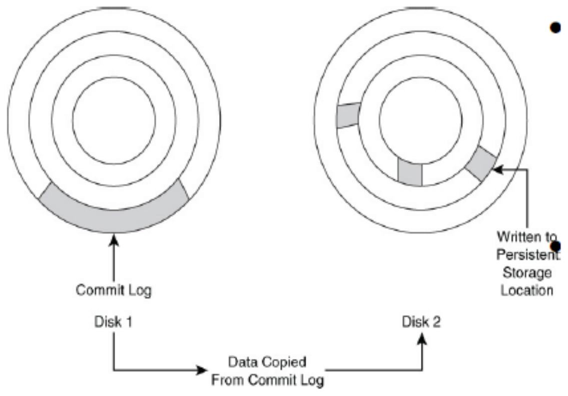
Cassandra peer-to-peer architecture

We can consider **different architecture** for *CD*; two of the most famous DBMSs which works with *CD*:

- **Apache HBase**: it exploits multiple type of nodes which together build the entire Hadoop environment (HBase is based on Hadoop)
  - **Master + Region Servers** regards the DBMS (based on Hadoop)
  - The rest regards the servers for building the framework itself
    - **Zookeeper**: server which is a node which enables the coordination between the different nodes of the Hadoop cluster. Its role is to mantain a shared hierarchical namespace

        - **Name Node + Data Node**: used for building the distributed filesystem of Hadoop and its replicas
    - The **point of strength** of this architecture regards mainly the fact that we have specialized software devoted to a specific task deployed on the different servers
    - The **cons** of working with this architecture is that we need to **take care** when **setting up** and **merging** it, which usually requires at least one specialized system administrator which manages the configuration and which tunes the different configurations of each server and each software
- **Cassandra**: exploits a **peer-to-peer** architecture with a **ring abstraction**
    - All the nodes run the **same software** but they can **different functions** for the cluster
    - Each node can receive both **read** and **write** requests operation from a client
    - **Advantages**:
        - **Simplicity**
        - **No node** can act as a **single point of failure**, if a node has a problem then the architecture itself will autoconfigure itself to make the system continue to work
        - It is **easy** to **scale-up/scale-down**, using for example a **consistent hashing** for relocating the values **automatically**

Commit Logs


Commit Log
Disk 1
Written to Persistent Storage Location
Disk 2
Data Copied From Commit Log

- Commit Logs are ***append only*** files that always ***write*** data to the end of the file.
- These files **allow** column databases to ***avoid waiting*** for the definitive writes on persistent supports.

- Whenever a write operation is sent to the database, is stored into a commit log. Then, the the data is updated into the specific server and disk block.
- After a failure, the ***recovery*** process reads the commit logs and writes entries to partitions.

Bloom Filters

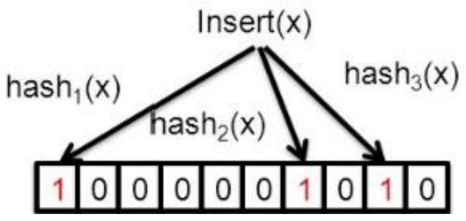
Bloom filters are ***probabilistic*** data structures. They help to ***reduce*** the number of ***reading operations***, avoiding reading partitions that certainly ***do not contain*** a specific searched data.

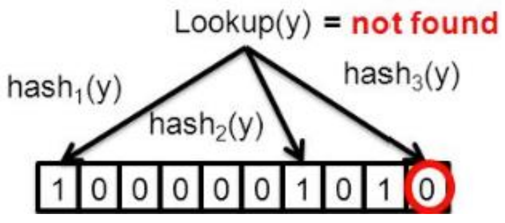
A Bloom filter tests whether or not an element is a member of a set: the answer of the test is "***no***" or "***maybe***". Thus, ***false positive*** may be obtained.
However, if the answer is "no" for sure the element is not present into the specific partition.

The same results may be obtained using ***hash functions***, even without false positive. However, ***bloom filters*** use ***less memory*** than hash functions.

An Example:

- Let consider a ***specific partition***. A bloom filter for this partition can be an ***array of n binary values***. At the beginning, all the elements are set to 0.
- Let consider also a set of ***k << n hash functions*** which map each data to be stored into the partition in an element of the array.
- When a new data is ***added (write)*** to the partition, all the hash functions are calculated and the corresponding bit are ***set to one***.
- When a ***read*** operation is required, all the has functions are calculated and the access to the partition is actually performed ***if and only if*** all the corresponding element in the array are ***set to 1***.
- The maximum false positive rate depends on the values of n and k.
- Let consider a Bloom Filter with n = 10 and k = 3.
    - The ***Insert*** function describes a ***write*** operation
    - The ***Lookup*** function describes a ***read*** operation





Consistency Level

Consistency level refers to the consistency ***between copies*** of data on ***different replicas***.

We can consider different level of consistency:

- ***Strict consistency***: all replicas must have the same data
- ***Low consistency***: the data must be written in at least one replica
- ***Moderate consistency***: all the intermediate solutions

The consistency level depends on the specific ***application requirements.***

Replication

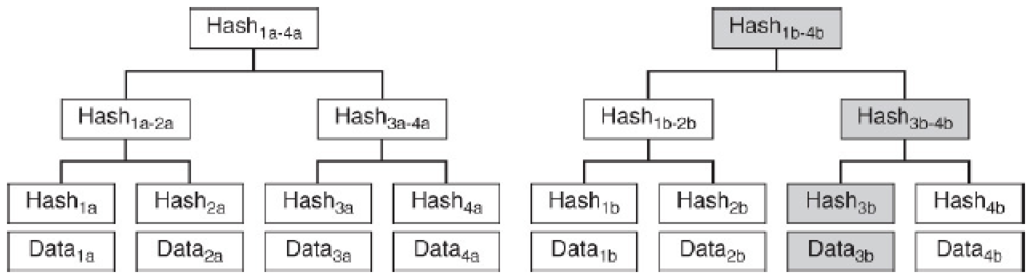
- Consistency level is strictly related to the ***replication process***.
- Replication process regards where ***place*** replicas and how to ***keep*** them ***updated***.
- Usually, there is a pre-defined server for the ***first replica*** which is in charge of identifying the position of the other replicas among the available servers.
- The primary server is often determined by a ***hash function***.
- Each database has its own ***replication strategies***.
- For example, Cassandra uses clusters organized in ***logical rings*** and additional replicas are stored in successive nodes in the rings in ***clockwise directions***.

Anti-Entropy

Anti-entropy is the process of ***detecting differences*** in replicas.

It is important to detect and resolve inconsistencies among replicas ***exchanging a low amount*** of data.

Hash trees may be exchanged among servers in order to easily check differences. Each leaf of the tree contains the hash value of a data set in the partition. Each parent node contains the hash value of the hashes its child nodes.



The server receiving the hash tree calculates its own hash tree and starts **comparing** it with the one received **starting** from the root:

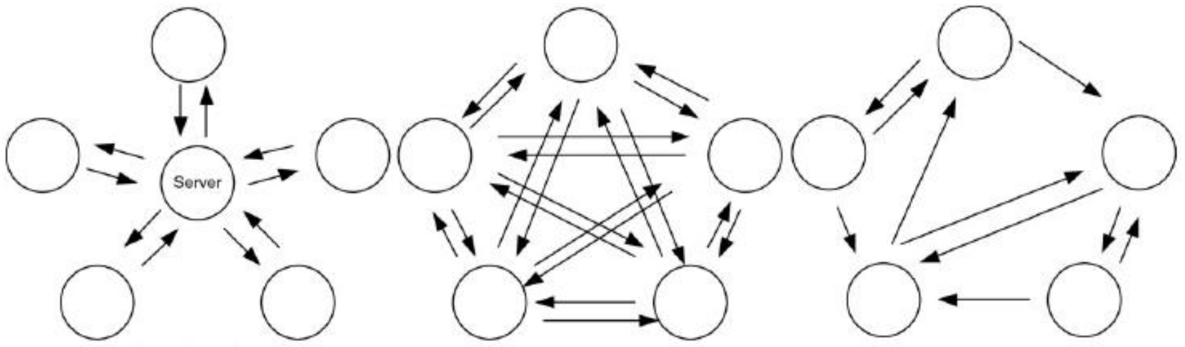
- If the root are the same, then **the comparison ends without detecting differencies**
- If they are not the same, then at least one of its child is different respect to the corresponding child in the other tree
    - Then, the root's child are compared with the same procedure until it arrives to the leaves (**data**), finding the defecting data

Communication Protocols

In distributed systems, each node needs to be ***aware*** about the ***status*** of the other nodes.

In small cluster, each node can ***exchange messages*** with each other node.

The number of total ***messages to exchange*** for a complete communication is equal to ***n×(n–1)/2***, where *n* is the number of nodes of a cluster.



(a) Centralized approach, which uses a centralized server for forwarding and mixing of multimedia streams

(b) Fully connected overlay approach, where all peers are in direct contact with one another

(c) Gossip-based approach, where peers operate in parallel, and each peer communicates with one or more randomly selected partner

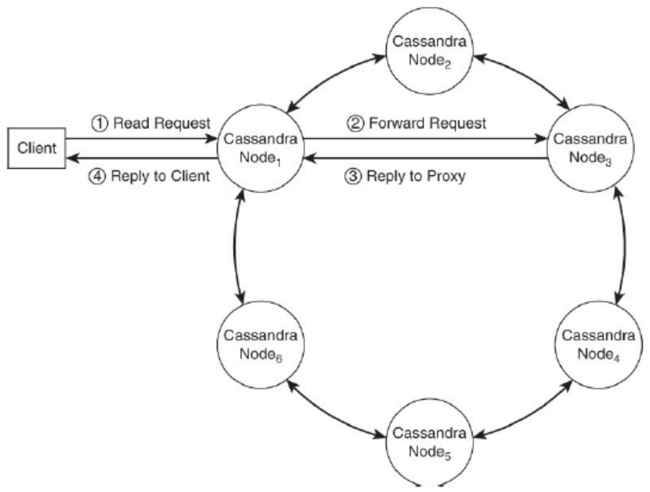
In the **gossip protocol** each node sends informations regarding itself and regarding all the nodes from which it received informations.

Cassandra

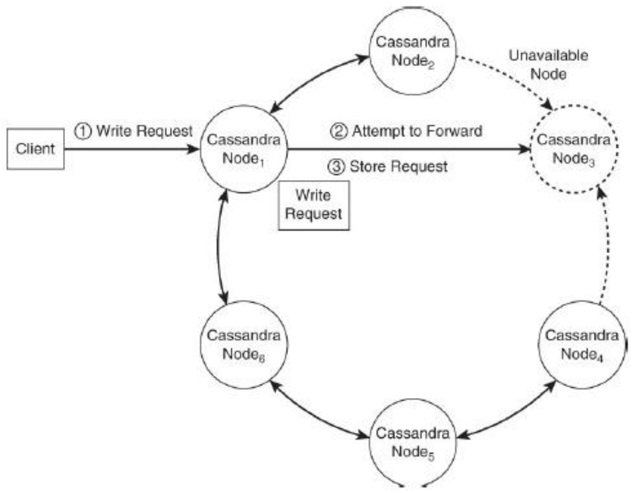
Cassandra uses the gossip-based protocol.

Any node in a Cassandra cluster can handle both read and write requests and, thanks to the gossip protocol, each node has information about the other nodes of the entire cluster (just asking to the nodes it is connected with about the state of nodes connected to them) and can act as a **Proxy** for clients.

Cassandra Read Protocol:



Cassandra Write Protocol:



In case the node interested in the write operation is down, we can use the **Hinted Handoff**.

Hintend Handoff

What happens if the node interested in a write operation is down?
Two scenarios may happen:

1. The operation will be lost, thus the target node will be in inconsistent state after its recover
2. A specific node can be designated to receive write operations and forward them to the target node when it becomes available.

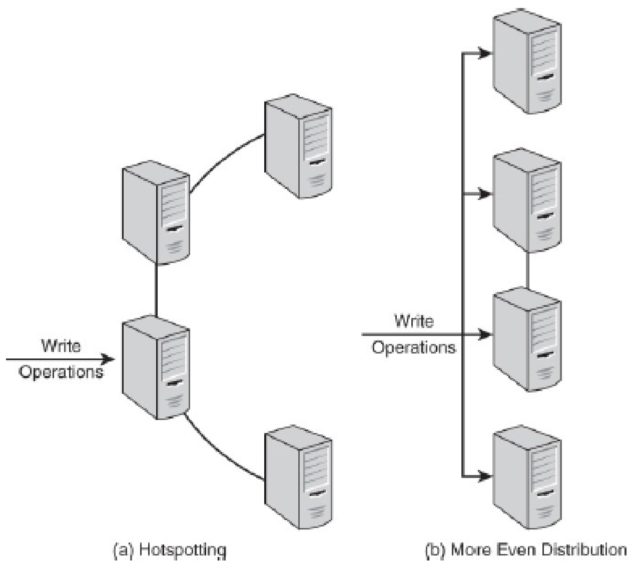
The hinted handoff ***process*** is in charge of:

1. ***Creating*** a data structure to store information about the write operation and where it should ultimately be sent
2. Periodically ***checking*** the status of the target server
3. ***Sending*** the write operation when the target is available

Once the write data is ***successfully written*** to the target node, it can be considered a successful write for the purposes of ***consistency*** and ***replication***.

Avoid Hotspotting in Row Keys

Hotspotting occurs when many operations are performed on a small number of servers



We can prevent hotspotting by ***hashing sequential values*** generated by other systems.

Alternatively, we could ***add a random string*** as a prefix to the sequential value.

These strategies would ***eliminate*** the effects of the ***lexicographic orde***r of the source file on the data load process.

p

INDEXING

In column databases, we can **look up** a **column value** to quickly **find rows** that reference that column value.

Primary Indexes are indexes on the **row keys** of a table (created and mantained by the DBMS).

Secondary Indexes are indexes created on one or more **column value**. Either the database system or my application can create and manage secondary indexes.

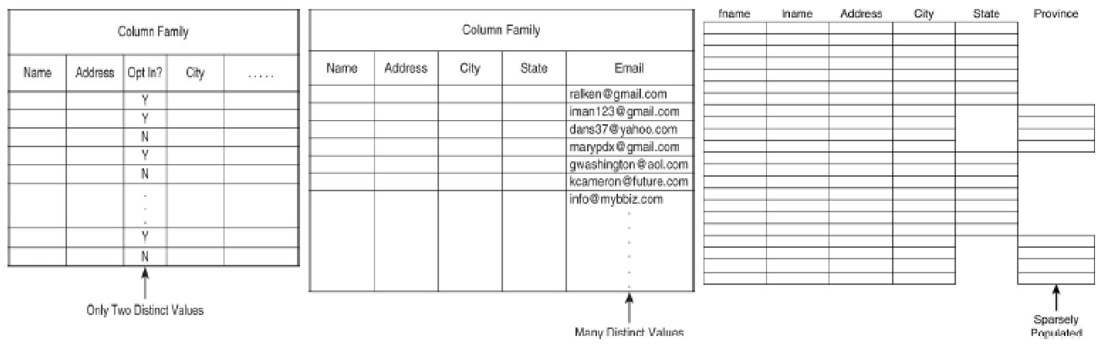
Secondary Indexes Managed by the Database Management System

General and commonsense rule: *"if we need secondary indexes on column values and the column family database system provides automatically managed secondary indexes, then we should use them."*

Example: we can speedup this query
SELECT fname, lname FROM customers WHERE state = 'OR' AND lname = 'Smith'
By setting an index on the *state* and on the *lname* attributes.
Typically, the DBMS will use the ***most selective*** index first.

The automatic use of secondary indexes has the advantage the we do not have to change our code to use the indexes, if the application requirements will change.

When avoiding to use automatically managed indexes



An index will probably not help much, especially if there are ***roughly equal numbers*** of each value.

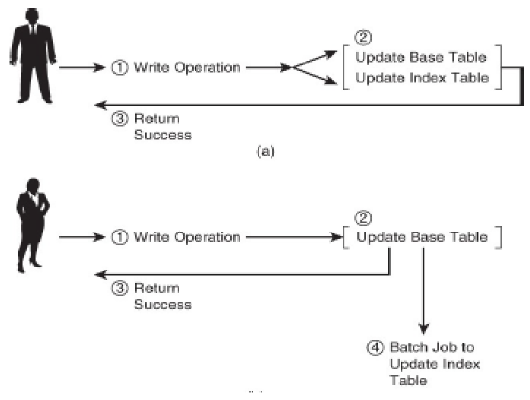
Indexes may not help much here because the index will have to ***maintain so much data*** it could take ***too much time*** to search the index and retrieve the data.

Too few values associated to a specific column. It is not worth creating and managing an index data structure for this attribute.

Create and Manage Secondary Indexes Using Tables

In this case, we have to ***explicitly create*** and manage tables to store data we would like to access via the index.

The ***programmer*** will be be ***responsible*** for maintaining the indexes and two main strategies may be adopted:



a) ***Updating an index table during write*** operations keeps data synchronized but ***increases the time*** needed to complete a write operation.

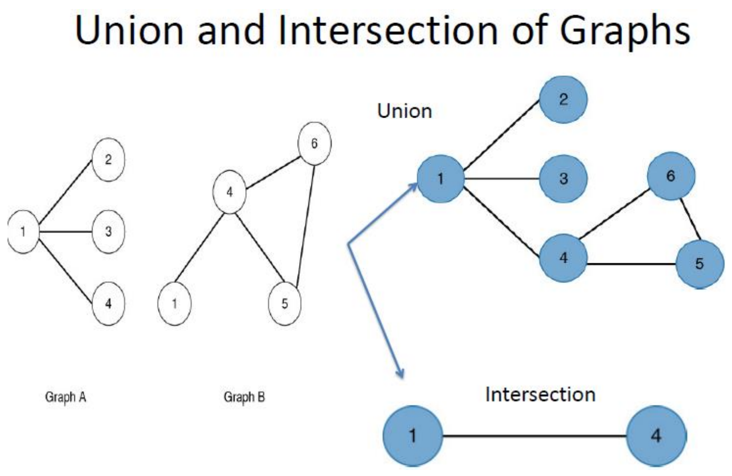
b) ***Batch updates*** introduce periods of time when the ***data is not synchronized***, but this may be acceptable in some cases.

GRAPH DATABASES (*GD*)

Each **Vertex** of the graph is an **instance** of an **entity** and can have multiple **properties** (attributes).

Edges are **relationships between vertices**; they can have **properties** associated with them, like a **weight**.

Graphs can be **directed** or **undirected** (have or not a **direction**), it depends on the **specific application**.

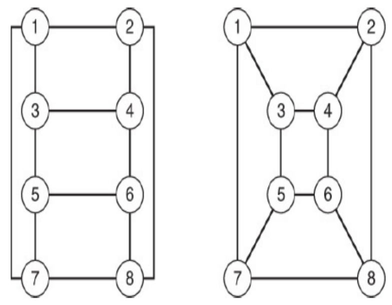


Graph Traversal: it is the process of **visiting all vertices** in a graph.

It's purpose is usually to either **set** or **read** some property value and is often carried out in a way such as **minimizing the cost** of the path.

Two graphs are **isomorphic** if:

- ***for each vertex*** in the first graph, there is a ***corresponding vertex*** in the other graph
- ***for each edge*** between a pair of vertices in the first graph, there is a ***corresponding edge*** between the corresponding vertices of the other graph

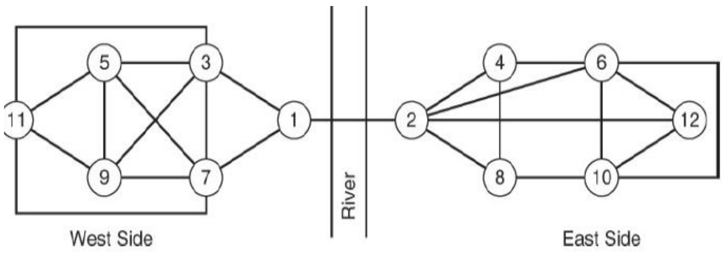


Detecting isomorphism in graphs or sub graphs allows us to identify ***useful patterns***.

Measures of a Graph

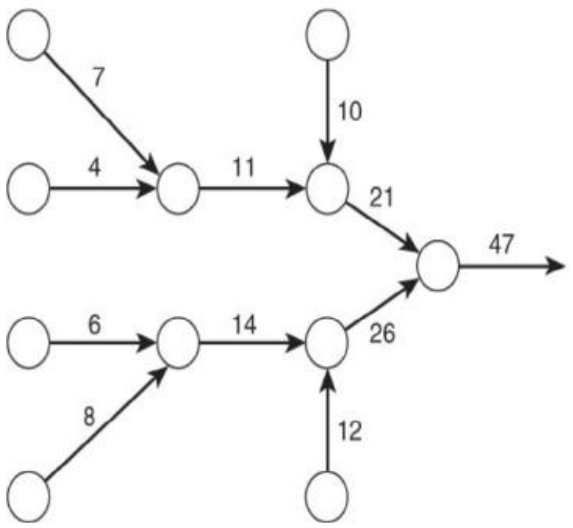
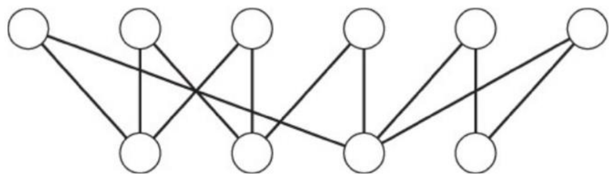
- **Order**: the number of vertices
- **Size**: the number of edges
  - These two measures are very important to **evaluate time** and **memory occupancy** required to handle specific operations; the **higher** the **order** and the **size** of the graph, the **harder** will be to solve those operations/queries.
- **Degree**: the number of edges linked to a vertex
  - It measures the **importance** of any given vertex in a graph

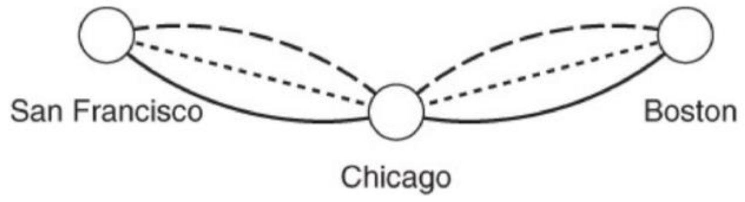
- **Closeness**: the measure of how far the vertex is from all the others in the graph. It can be calculated, in a fully connected graph, as the **reciprocal of the sum** of the length of the **shortest paths** between the node and all the other nodes in the graph
  - It is useful to **evaluate** the **speed of information** spreading in a network, starting from a specific node
- **Betweenness**: the measure of how much of a **bottleneck** a given vertex is



  - *In the example, if either node 1 or 2 is removed then the graph becomes disconnected*

<u>Type of Graph</u>

- **Flow Network**
  - a flow network is a **direct graph** which adopts **capacitated** edges
  - Each vertex has a set of **incoming** and **outcoming** edges
  - In each vertex the sum of the capacities of incoming edges **cannot be greater** than the sum of the capacities of outcoming edges
  - The unique exception to the previous rule regards the **source** and the **sink** nodes
  - 
- **Bipartite Graph**
  - There are two **distinct sets** of vertices
  - Each vertex is connected only with vertices of **the other set**
  - 

- **Multigraph**
  - It is characterized by the presence of **multiple edges** between vertices
  - Each edge can be equipped with **different sets of properties**
  - 

In **relational databases** we used **join** operations to find connections or links between entities; this operation can be **computationally expensive** with huge amount of data.
Instead, in *GD* the edges between entities allow us to **quickly** query the databases.

Edges allow us to **explicitly** modeling many-to-many relations rather than using tables.

Different type of **relations** between entities may be handled ecploiting **multiple** type of edges, which can be simply obtained specifying **different values** of edge **properties**.

Graph databases are **suitable** for problem domains easily described in terms of entities and relations.

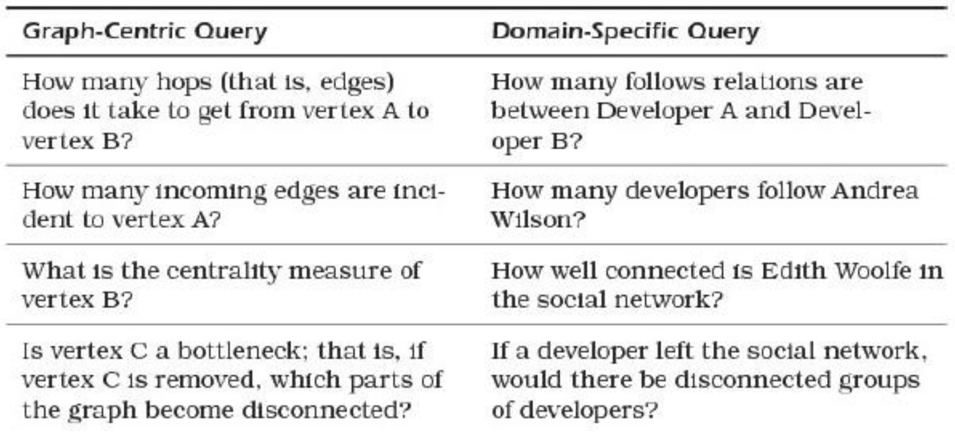
SQL lacks the syntax to easily perform **graph traversal** (especially when the depth is unknown or **unbounded**).
Performance degrades quickly as we traverse the graph; each **level of traversal** adds significantly response time to a specific query.

Analyzing Relations: instead of considering ***all the possible combinations*** for the identification of candidate relations, as the ***number of entities grows***, it is better to focus on combinations that are ***reasonably*** likely to ***support*** queries related to the ***application requirements***.

Mapping Queries

Queries drive the design of the graph databases.

Domain-specific queries must **be translated** into **Graphic-centric** queries, after the **identification** of **entities** and **relations** between entities



Graph-Centric Query	Domain-Specific Query
How many hops (that is, edges) does it take to get from vertex A to vertex B?	How many follows relations are between Developer A and Developer B?
How many incoming edges are incident to vertex A?	How many developers follow Andrea Wilson?
What is the centrality measure of vertex B?	How well connected is Edith Woolfe in the social network?
Is vertex C a bottleneck; that is, if vertex C is removed, which parts of the graph become disconnected?	If a developer left the social network, would there be disconnected groups of developers?

Advices

1. Pay attention to the ***dimension*** of the graph: the bigger the graph the more ***computational expensive*** will be performing queries.
2. If available, define and use ***indexes*** for improving retrieval time.
3. Use ***appropriate*** type of ***edges***:
    a. In case of symmetrical relations, use undirected edge, otherwise use direct edges.
    b. Properties requires memory! If possible, ***use simple edge*** without properties. Pay also attention to the data type of the properties.
4. ***Watch for cycles*** when traversing graphs: in order to avoid endless cycles, use appropriate data structure for ***keeping track*** of visited nodes (if not, we could find ourself in an endless cycle).

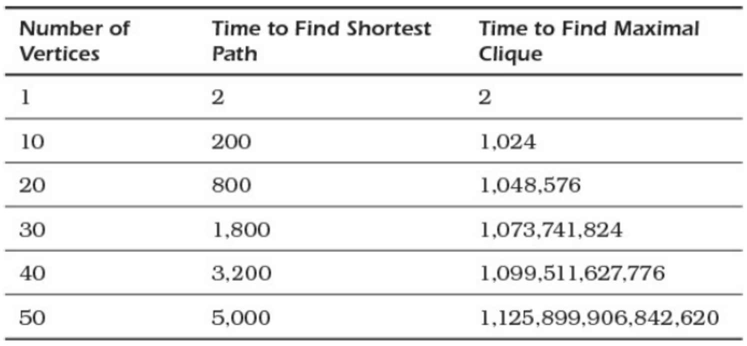
Scalability of Graph Databases

Most of the famous implementation of graph databases do not consider the deployment on cluster of servers (***no data partitions nor replicas***).

Indeed, they are designed for running on a ***single server*** and can be only ***vertically scaled***.

Thus, the developer must take a special attention to the growing of:

- The number of nodes and edges
- The number of users
- The number and size of properties



Number of Vertices	Time to Find Shortest Path	Time to Find Maximal Clique
1	2	2
10	200	1,024
20	800	1,048,576
30	1,800	1,073,741,824
40	3,200	1,099,511,627,776
50	5,000	1,125,899,906,842,620

*The **Maximal clique** is the largest "group of people who follow each other"*

IN-MEMORY DATABASE (*IMD*)

if our application ***does not need*** to handle huge amount of data and to scale, both vertically and horizontally, we can think about adopting ***different solutions*** that the ones discussed so far.

In-memory databases may be suitable for applications whose handled data may ***fit all*** in the memory of a single server.

Moreover, there also can be databases that can reside in the memory capacity of a cluster (***if replication is needed***).

In-Memory databases usage is been boosted by the **decreasing** price of *GB of memory* and by the **increasing** *MB per Chip* ratio.

Features

- Very ***fast access*** to the data, avoiding the bottleneck of the I/O transfers.
- Suitable for application that do not need ***to write continuously*** to a persistent medium.
- ***Ad-hoc architecture*** for being aware that its data are resident in memory and for ***avoiding the data loss*** in the event of a power failure (***alternative persistency model***).
- ***Cache-less architecture***: despite traditional disk-based databases, a memory caching systems is not need (every thing is already in memory!!!).

<u>Solutions for Data Persistency</u>

In the following, we show some solutions for avoiding data loss in case of ***system failure***:

- ***Replicating data*** to other members of a cluster.
- Writing complete database ***images*** (called snapshots or checkpoints) to disk files.
- Writing out transaction/operation records to an append-only disk file (called a ***transaction log*** or ***journal***).

<u>TimesTen</u> (Oracle Solution)

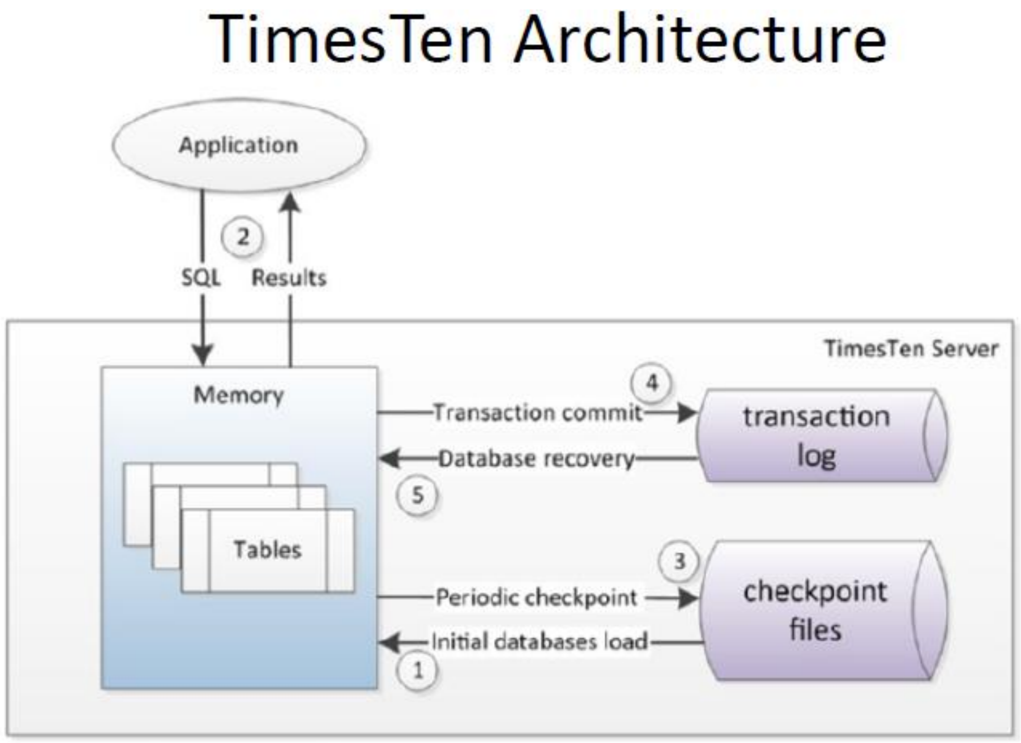
TimesTen implements a fairly familiar ***SQL-based*** relational model.

All data is ***memory resident.***

Persistence is achieved by writing:

- ***periodic snapshots*** of memory to disk
- ***transaction log*** following transaction commits.

By default, the disk writes are asynchronous! To ensure ACID transactions the database can be configured for synchronous writes to a transaction log after each write (performance loss).



Redis

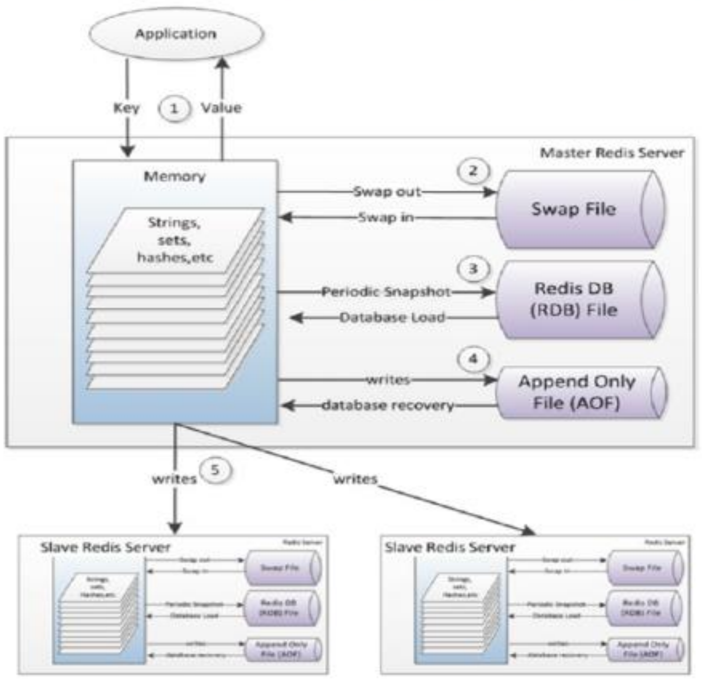
Redis is an in-memory ***key-value*** store.

Objects consist mainly of strings and various types of collections of strings, such as ***lists*** and ***hash maps***.

Only ***primary key*** lookups are supported (***no secondary indexing*** mechanism allowed).

Redis may exploit a sort of "***virtual memory***" mechanism for storing an amount of data larger than the server memory. Old keys are ***swapped out*** to the disk and recovered if needed (***lost of performance***).

Redis: Architecture



Redis supports asynchronous master/slave ***replication***.

If ***performance is very critical*** and ***some data loss*** is acceptable, then a replica can be used as a backup database

In this case, the master is configured with ***minimal disk-based persistence***.

Redis may replicate the state of the master database asynchronously to slave Redis servers.

HANA DB (a SAP product)

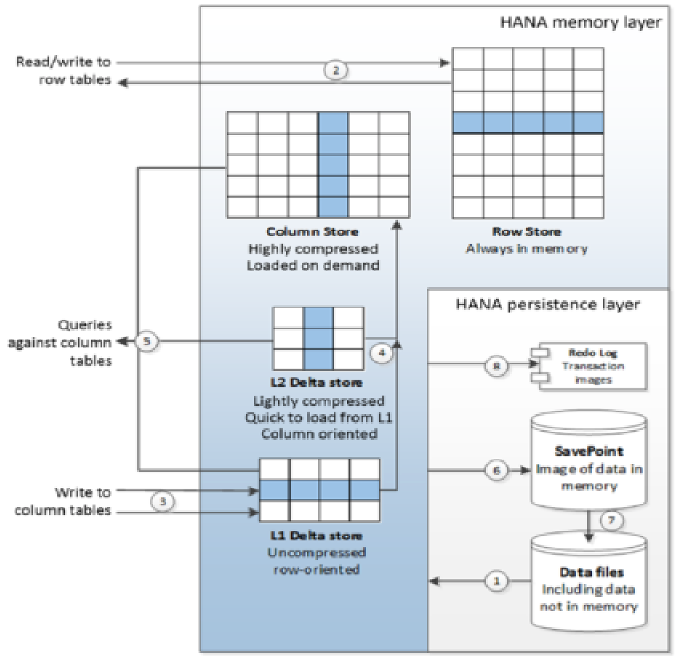
SAP HANA is ***a relational database*** which combines an ***in-memory*** technology with a ***columnar*** storage solution.

It must be installed on an ***optimized hardware configuration***.

Classical ***row tables*** are exploited for ***OLTP*** operations and are stored ***in memory***.

Columnar tables are often used for ***OLAP*** operations. By default, they are ***loaded in memory by demand***. However, the database may be configured to load in memory a set of columns or entire tables since the beginning.

HANA DB: Architecture



Persistence:

- Images of memory are copied to save points periodically (6)-
- The save points are merged with the data files in due course (7).
- When a commit occurs, a transaction record is written to the redo log (8) (on fast SSD).

<u>Oracle 12c</u>

It is mainly a **relational model** in which data are **stored** in **row** fashion.

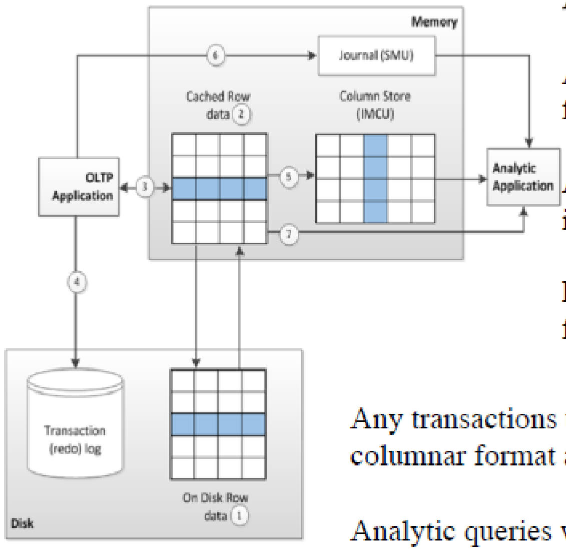
Oracle 12c: Architecture

Data is maintained in disk files (1), but ***cached in memory*** (2).

An *OLTP* application primarily reads and writes from memory (3),

Any ***committed transactions*** are written immediately to the transaction log on disk (4).

Row data is loaded into a ***columnar representation*** for use by ***analytic applications*** (5).



Any transactions that are committed once the data is loaded into columnar format are ***recorded in a journal*** (6),

Analytic queries will ***consult the journal*** to determine if they need to read updated data from the row store (7) or to rebuild the columnar structure.

GUIDELINES FOR SELECTING A DATABASE

Whenever we are asked to design and develop a specific software application, aset of steps must be performed:

1. ***Requirements elicitation*** from customer
2. ***Requirements definition***, both functional and non functional
3. ***Use case*** definition (better if with UML diagrams)
4. Identification of the main data structures (including the main procedures) and of the relations among them (***Analysis Workflow***)
5. Refinement of 4) (***Project Workflow, including DB Design***)
6. ***Implementation*** and ***test***
7. ***Deploy***

The **role** of the architect/engineer is to identify the **most suitable**:

1. Software/Hardware ***architecture***
2. ***Programming languages*** and development environments
3. ***Database management systems***

Everything must be ***driven*** by:

1. Requirements
2. ***Common sense***
3. Experience

Are Relational Databases still useful?
Yes, of course! They do the work of **protecting** the **data integrity** and **reducing** the **risk of anomalies** (*immediate consistency* and *ACID transactions support*):

- Relational databases ***will continue*** to support ***transaction*** processing systems and, in general, ***OTPL*** applications
- ***Decades of experience*** with transaction processing systems and data warehouses has led to ***best practices*** and design principles that continue to meet the needs of businesses, governments, and other organizations

The Modern Application World

Modern data management infrastructure is responsible for a ***wider range*** of applications and data types ***than ever before***.
We are experiencing ***the era of***:

1) Big Data
2) Internet of Things
3) Large scale web-application
4) Mobile application

In this new era, often ***performance*** and ***availability*** are more important than consistency and ACID transactions!

On the basis of our requirements, analysis and project workflows, we can choice among:

- ***Relational databases***, such as PostgreSQL and MySQL,
- ***Key-value databases***, such as Redis, Riak, and DinamoDB
- ***Document databases***, such as MongoDB and CouchDB
- ***Column family databases***, such as Cassandra and HBase
- ***Graph databases***, such as Neo4j and Titan

The choice has to be guided by the requirements (**requirement driven**); the ***functional requirements*** are related to the type of queries that describe how the data will be used. Moreover, they also influence the ***nature of relations*** among entities and the ***needs of flexibility*** in data models.

When choosing a database we have to analyze other factors (***non functional requirements***) such as:

- ***The volume*** of reads and writes
- ***Tolerance*** for inconsistent data in replicas
- ***Availability*** and ***disaster recovery*** requirements
- ***Latency*** requirements

Key-Value Databases

These databases are preferable when:

- The application has to handle ***frequent but small*** read and write operations.
- ***Simple data*** models have been designed
- ***Simple queries*** are needed (no complex queries and filtering on different fields)

These databases do not allow too much flexibility and the possibility of making queries on different attributes.

In the following, we show some examples of applications that may exploit these databases:

- ***Caching data*** from relational databases to improve performance
- ***Tracking*** transient attributes in a web application, such as a ***shopping cart***
- ***Storing configuration*** and user data ***information*** for mobile applications
- ***Storing large objects***, such as images and audio files

<u>Document Databases</u>

These type of databases are suitable for applications that need:

- To store ***varying attributes***, in terms of number and type
- To store and ***elaborate large amount of data***
- To make ***complex queries*** (on different types of attribute), to use ***indexing*** and to make ***advanced filtering*** on attributes.

Flexibility is the main feature of document databases, along with high ***performance capabilities*** and ***easy of use***.

Embedded documents allow to use few collections of documents that store together attributes frequently accessed together (***denormalization***).

In the following, we show some examples of applications that may exploit these databases:

- Back-end support for websites with high volumes of reads and writes
- Managing data types with ***variable attributes***, such as products
- Tracking ***variable types*** of metadata
- Applications that use ***JSON*** data structures
- Applications benefiting from ***denormalization*** by embedding structures within structures

<u>Column Databases</u>

These type of databases are suitable for applications that:

- Require the ability to ***frequently read/write*** to the database
- Are ***geographically distributed*** over multiple data centers
- Can tolerate some ***short-term inconsistency*** in replicas
- Manage ***dynamic fields***
- Actually have to deal with ***huge amoun****t* of data, such as hundreds of terabytes

Column Databases are normally deployed on clusters of multiple servers on different data centers. If data handled by the application is small enough to run with a single server, then avoid these databases.

In the following, we show some examples of applications that may exploit these databases:

- ***Security analytics*** using network traffic and log data mode
- ***Big Science***, such as bioinformatics using genetic and proteomic data
- ***Stock market analysis*** using trade data
- ***Web-scale applications*** such as search
- ***Social network*** services

Graph Databases

These type of databases are suitable for applications that:

- Need to represent their domain as ***networks*** of connected entities
- Handle instances of entities that have ***relations*** to other instances of entities
- Require to ***rapidly traverse*** paths between entities.

Some considerations:

- Two ***orders in an e-commerce*** application probably have no connection to each other. They might be ordered by the same customer, but that is a shared attribute, not a connection.
- A ***game*** player's configuration and game state have little to do with other game players' configurations. Entities like these are readily modeled with key-value, document, or relational databases.
- Let consider ***proteins*** interacting with other proteins or ***employees*** working with other employees. In all of these cases, there is some type of connection, link, or direct relationship between two instances of entities.

In the following, we show some examples of applications that may exploit these databases:

- Network and IT ***infrastructure management***
- ***Identity*** and ***access*** management
- ***Business process*** management
- ***Recommending*** products and services
- ***Social networking***

Let's Cooperate!

When choosing a solutions for data storage and management, we are ***not obliged*** to use ***just one type*** of database management systems.

Since different types of SQL and NoSQL DBMSs have different features, that can be usefully together in a specific application, they may be used ***concurrently***.

Some use cases:

- ***Large-scale graph processing***, such as with large social networks, may actually use ***column family databases*** for storage and retrieval. Graph operations are built on top with a graph in-memory database.
- ***OLTP*** handled with a Relational Database (for ACID transactions), ***OLAP*** in charge to MongoDB for fast analytics.