LARGE SCALE AND MULTI STRUCTURED DATABASES

green text: whole big pieces of text taken directly from the slides because i found them useful

Introduction to the Course

The Big Data Era

Big data is a fashion trend buzz word, but it is being used everywhere and by everybody. It is a reality we are living in and working with in the information technology world.
Many people say that they work with big data but they do not say what they can do with it.
The question is: what can we do with Big Data? How can we use Big Data?
We already know that the amount of data produced every minute accumulates and grows in real time. So, big data exists, we want to know what to do with it.

What are the sources of big data?
Social media and networks, transactions from banks etc, scientific instruments like satellites, sensors and beacons, mobile devices, sensor technology and networks, genomics.

Usually, big data are represented with V's, there are many but these are the main five:

- Volume - the amount of data produced
- Variety - data is produced from different sources and in different formats
- Velocity - data is produced at a really high speed
- Veracity - security of the data, it could be manipulated and just not true
- Value - data must permit us to extract useful information from it

There are also other 4, which are additional features:

- Validity - data quality
- Variability, has to do with variety and velocity
- Vocabulary - structure of the data as in data models and semantics
- Vagueness, which is the confusion over the meaning of big data and tool used

In reality, Vs are 42 but… whatever

Big data has many faces and aspects. it is not only needed for analytics and prediction - there are many aspects to take care of.
Faces include:

- Data acquisition - it is not easy to design systems for this but the course will not talk about this
- Storage infrastructure
- Computation infrastructure
- Databases/querying

- *Analytics/mining*
- Visualization - a big challenge, which may involve mixed and augmented reality and needs creativity
- Security and privacy - this is very important

Main big data issue: 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.
In other words, that much data renders normal systems unusable. We must change paradigm.

What about computing?
Classical data elaboration algorithms, such as the ones for analytics and data mining, cannot 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.

In the past, when data increased, we scaled up: this means, we got bigger servers with more memory and bigger disks. It is very expensive. One of the first companies which had to manage these costs was Google.
The idea they had was to add more servers with the same capabilities, and build clusters. This is called scaling out.
The economic problem works in parallel with the computing paradigm that we need to exploit: Distributed Computing.
The idea behind it is to have a big amount of data and split it in different servers which form a cluster.
Later in the course we will see that partitioned data will be distributed and managed automatically by DBMSs. Splitting data, or sharding, can be done in different ways, which might require replication of the shards to ensure availability.

Let's go back to computing. The idea is: put the data close to the computing unit. Of course, we must reduce the amount of data clustered among the computing unit, to reduce overhead.
Data locality means that we want to let each computer work on its local data. Since one machine cannot process or store all data, it is distributed in a cluster of computing nodes. It does not matter which machine executes the operation nor if it is run twice in different nodes (due to failures or straggler nodes).
We look for an abstraction of the complexity behind distributed systems
– DATA LOCALITY is crucial, just like to avoid data transfers between machines as much as possible.

Data can be replicated: we can have algorithms run on the same data on more than one machine, not a problem.

Let's imagine having the space that we need, and that data is replicated in the different disks of the different machines. computing is performed on the data, by all the computers, in

parallel. the communication among the servers and the nodes will only interest the results achieved. each of the computing units can dedicate time to chunks of data.

We store data on local disks, perform computations locally, and share results between units. It makes no sense to have one big storage. We should avoid sharing data between the units.

New programming model: MapReduce
"Moving computation is cheaper than moving computation and data at the same time"
It is a programming model, not so new because it existed before big data era.
Idea:

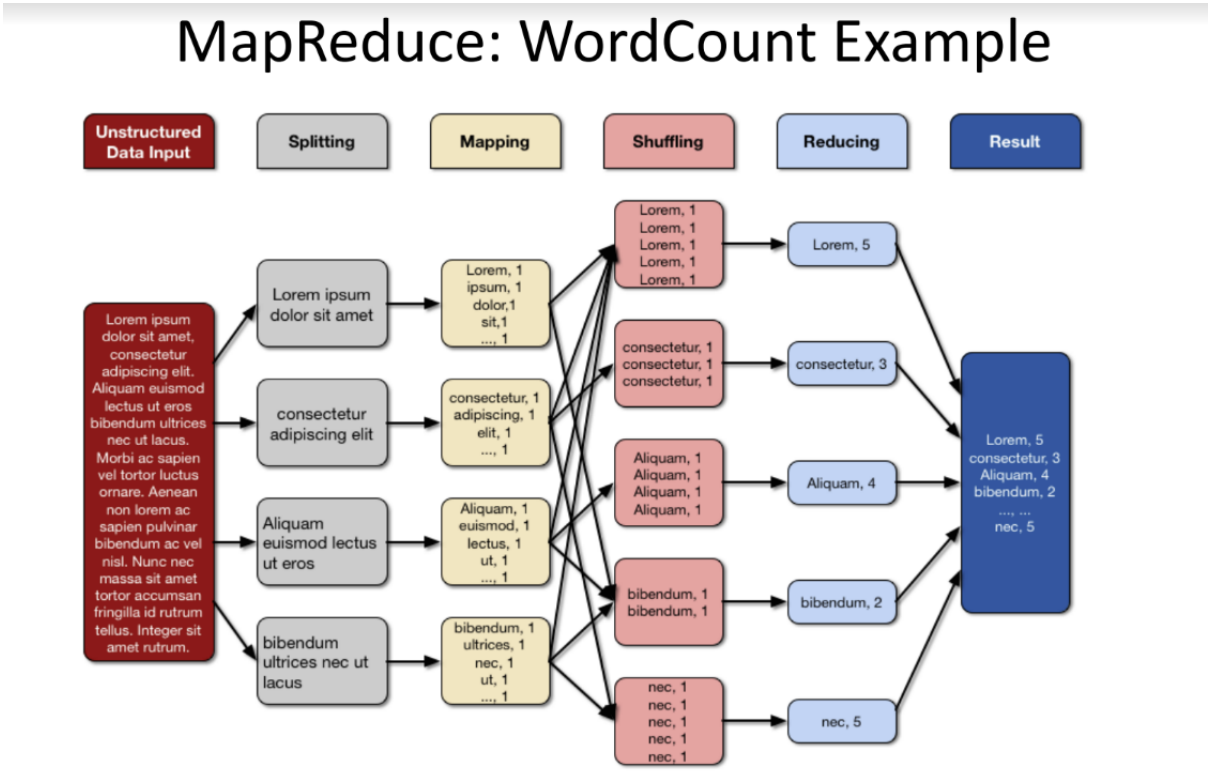
- Data is distributed among nodes (distributed file system)
- Functions/operations to process data are distributed to all the computing nodes
- Each computing node works with the data stored in it
- Only the necessary data is moved across the network

We have two components, the mapper and the reducer.
Mapper: a program which runs on a computing unit and works on a chunk of data. Each mapper does the same thing in a specific contest.
Given the data, we divide it into chunks and each one is executed by one mapper.
Then the mapper is run on a computing unit and one of those can run several mappers on each chunk of data.
Then we have the reducer: a program that will run on one or more computing unit and will aggregate the results produced by the mapper.

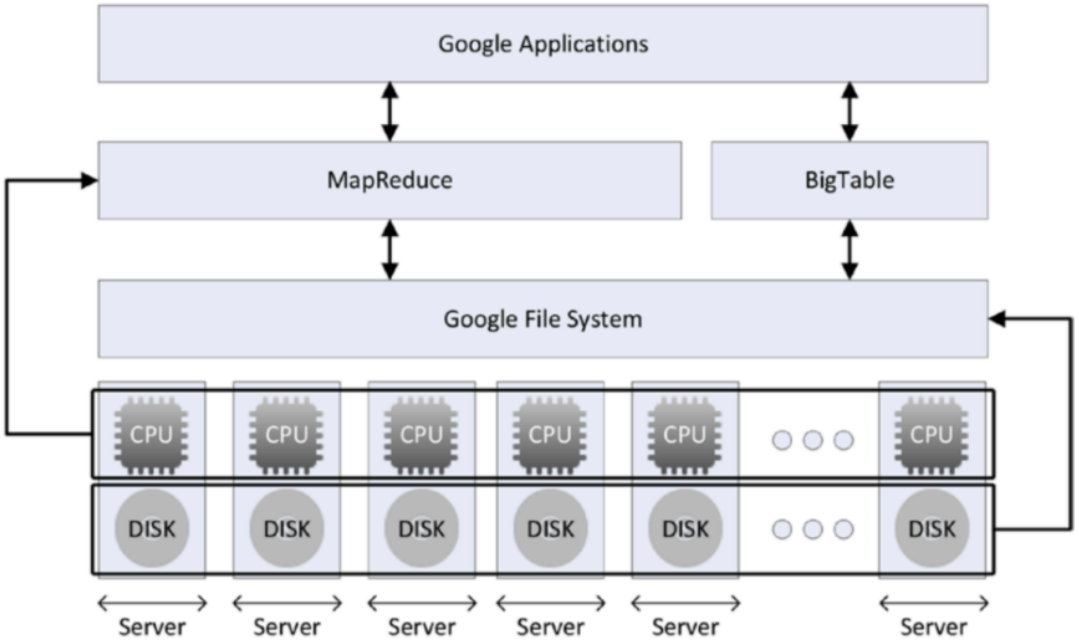
It is a parallel programming model introduced in 2004 by Google, which makes use of the Divide et Impera strategy, where Divide is the Mapper (partition dataset into smaller, independent chunks to be processed in parallel) and Impera is the Reducer (combine, merge or otherwise aggregate the results from the previous step).

How does it work? Let's take a very quick look. Suppose that we have the input data and that we divide it into multiple logical splits. These logical splits are associated each to one mapper.
We have a mapper running on one computing unit. If, for example, we have three mappers, we have three computing units. The scheduler of the operating system which considers to have three computing units, the different tasks associated with the mappers will be managed by the scheduler - at a very high level we see that we have a mapper for computing unit. Each mapper will produce results, which will be sorted and organized in order to be elaborated by the reducer. The reducer, again, is formed by tasks that will be managed by the scheduler of the operating system (and we have a framework on the operative system) so the reducer will aggregate the results and produce the final results.

The basic working: imagine that we have one chunk of the data. The mapper works by taking an input (key, value) and produces another (key, value) output. This means that we have each record associated with a key and a value, and the mapper elaborates it and comes up with another key and value. Then, all the key and values are organized, sorted. We generate a collection of (k, v) pairs where we have a list of values as a value. Reducers work with a specific key. The output is a value.



The example in the picture counts the number of times of occurrence of each word.
We split the document into chunks, then the mapper gives a key equal to the word, and a value equal to 1 for each word.
Then, all the k, v couples are sorted by key (so we aggregate the same words). The reducer just counts the number of equal words and produces the result.



Google Software Architecture for Big Data:

Google experienced quite early the problems of big data because of the fame of the search engine, and they tried to solve the problem with a first approach to a big data architecture. The physical infrastructure is a simple battery of simple computing units, together with disks. What they did was a distributed file system.
A file system is a collection of data structures and methods that allows us to abstract the files and folders' structure. a distributed file system abstracts a collection of files that are contained in different disks.
So, this file system is the Google File System and allows us to distribute, even with replicas, and it is fault tolerant.
On the basis of this they used the MapReduce strategy based on a framework called Hadoop (the free version). The distributed DBMS that they use is BigTable. IT is a column based db.
This is the engine for the data storage on top of which we have Google Applications, at the higher level.
This was the first architecture. Many other services followed, like Amazon and Facebook.

Hadoop is a Google Framework and an Apache Project. At a file system level, this framework is used for storage of very large datasets and it supports distributed processing of very large datasets.
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

A few words on HDFS: it's written in Java, and it is a distributed file system that allows us to abstract the files for big data. It scales to several thousands computing nodes and each node stores a part of the dataset.
Fault tolerant due to data replication, designed for big files and low-cost hardware like GBs, TBs, PBs. Efficient for read and append operations (random updates are rare)

Its limits: when we run a program on hadoop, we must move the data from the file system to the memory and then, for each iteration, we must push back data from the memory to the file system. For example, if we must iterate several MapReduce instances, for each stage Hadoop reloads the data into the main memory from the file system.
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.

Another Apache framework is called SPARK. It's open source and still very used for big data. Hadoop and Spark have their own environments on systems that run around them, and they are strictly connected between them. Spark exploits HDFS. Data are loaded in the memory during the execution of the program in SPARK.
Spark also allows us to work with other kinds of paradigms and there is a thick environment around it.
Apache Spark is an open-source which has emerged as the next generation big data processing tool due to its enhanced flexibility and efficiency.

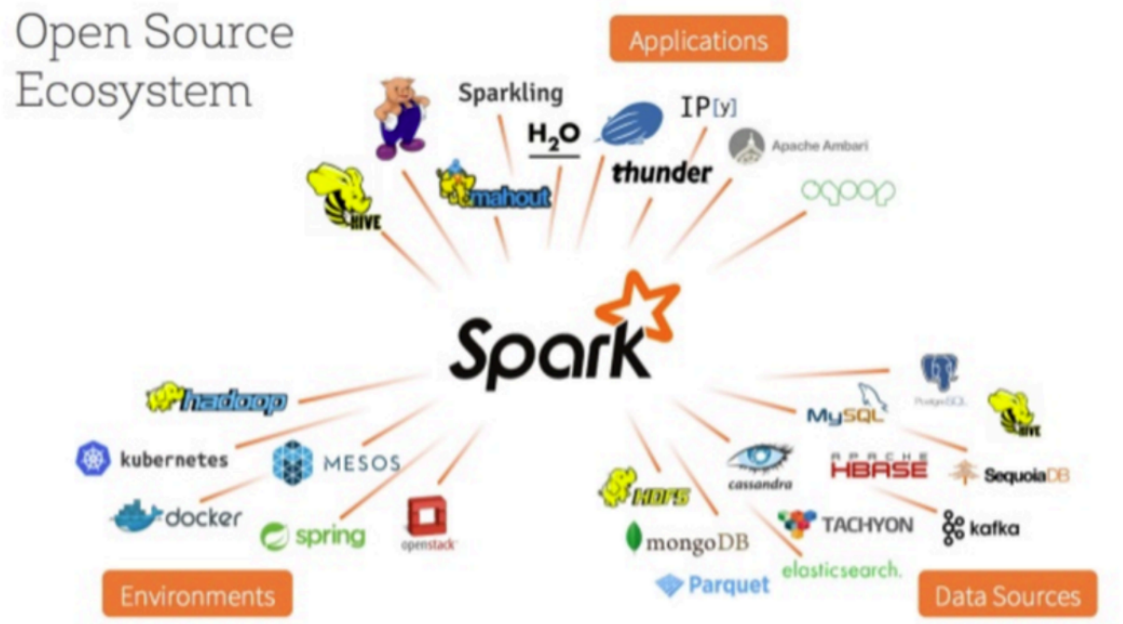
Spark allows employing different distributed programming models, such as MapReduce and Pregel, and has proved to perform faster than Hadoop, especially in case of iterative and online applications.
Unlike the disk-based MapReduce paradigm supported by Hadoop,Spark employs the concept of in-memory cluster computing, where datasets are cached in memory to reduce their access latency.
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.

We continue to have the same architecture as before, with many computers working on their local data. We have a driver program on one master machine, and different tasks distributed to the workers in the clusters. The workers can run, even in parallel, the different tasks.
The driver program runs the main functionalities, then the tasks are sent to the executor nodes.

The point of strength of Spark is the RDD (resilient distributed dataset), an abstraction which provides a representation of the data in memory. It is a dataset, it is distributed, it is resilient because we can rebuild it if we lose a piece of it. We use it for memory representation. If we look at a program written in Spark we see that we first load data, maybe from the HDFS, then we create the RDD, and then we work on the RDD. So, SPARK is independent from the source of the data.

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 few years, Data Frames and Datasets have been released as an abstraction on top of the RDD.

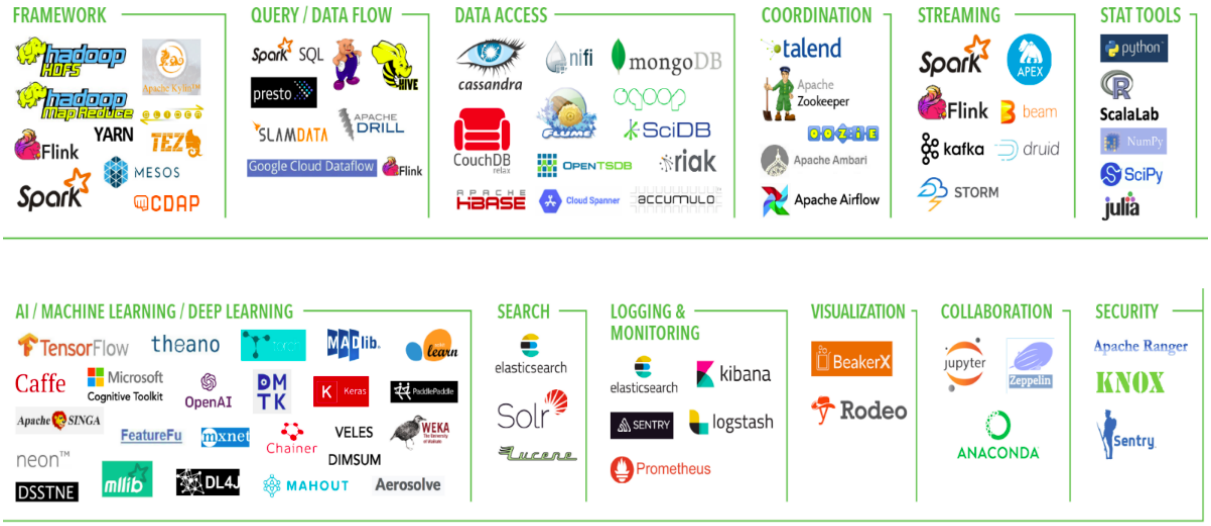
Spark Ecosystem:



This is the ecosystem around Spark - there are environments in which Spark can work. Spark can interact with many data sources, since It has several connectors towards the data, which are in charge of transforming data into RDD, dataframes and datasets.
There are also several applications which run on Spark.

Spark is up to 10x faster than Hadoop on disk, 100x in memory, and 2-5x less code is needed.

There are several open source solutions for working with big data.



Here, following, are some examples of which kind of problems can be solved via the usage of big data.

The first example that we see regards large-scale graph partitioning.

A graph is a collection of edges and vertices. Edges are relationships, vertices are entities. There may also be some properties described.
If we have a very big graph, like that of a social network, organizing it should be very important, for example in order to partition it.
We could have an optimal partitioning to lower the cost of graph queries.
(23:05)

Fundamentals and properties of the NoSQL databases

The Data Bases Revolution

The “prehistoric” database - why do we say this? Because a database is an organized collection of data. This means that a database doesn’t need to be digital. Even a dictionary, or an encyclopedia, is a database - made of paper, of course. Libraries and other indexed archives of information are also databases - they contain data, in a collection, and the data is organized!
In the 19th century, there were some tools that could be utilized to be databases. One example are loom cards, to create complex fabric patterns. Inside the wooden structures of the loom there was the pattern to be woven into the fabric.
Another example is the perforated paper strips for musical boxes. This contains organized information!
Another example: punched cards used in tabulating machines for producing census statistics to be stored and analyzed automatically. These are mechanical databases.
The “zero” revolution is the tape revolution: after the second world war, data was stored sequentially on tape - first paper and then tape.
Strength: it was easy and fast to fast forward and rewind.
Weakness: no direct high speed access to an individual record.

When did the first electronic databases appear? around the 1960’s there were the first mainframes, used to record the transactions for banks, stock prices etc. These were called OLTP (OnLine Transaction Processing). They are the first electronic databases. We have the first computer, and the databases were stored on the file system. So, we have a database but no DBMS.
ISAM (Index Sequential Access Method) system: like in the tape, data is organized into records of fixed length and all one after the other. There are also other records with the indexes that can be used as an associative array.
What is a DBMS: it is a software in charge of managing the database in terms of rules, consistency and other things it needs.
So, in order to solve the problems of not having a specific application for the data layer, all the methods used for managing the data should be written into the application. every application had to reinvent the database wheel.
Problems:

- Having to rewrite everything all the time might mean more frequent errors

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

The first database revolution was when we added a layer between data and the logical part of the application: it is the appearance of DBMS. Programmer overhead was minimized, we had a better performance of the database, integrity and performance were more insured than before. We started moving towards a certain organization and normalization of the data. DBMS needs the scheme and organization of the data to work properly, as well as the access path to move between information. So we need to define both of those elements.

The first kind of DBMS did not manage relational databases but something called navigational models.
We organize the data in tables, and link the tables between them. There were hierarchical models, where going from a place to another the paths and direction were constrained heavily. Network models might have some more freedom, but not necessarily.
Actually, this old databases are not very far from nosql databases, since in those last ones we must think in advance which will be the main queries and requirements of our application. In relational databases we have rules and a more strict organization with the normal form - these are good things. On the bad side, we have schemes and joins to handle.

Main issues of the navigational models:
- they practically only run on IBM mainframe systems
- all the queries that could be anticipated during the initial phase were possible, and only those
- adding new data elements to an existing system was very difficult
- We have DBMS, but it was mainly focused on CRUD operations (Create, Read, Update, Delete) and more complex analytic queries required hard coding in the program.

So, we moved from mechanical databases (which can be seen as databases but only in a more general way) then we have the tapes on which to record the information, then the digital database, with the file system, then DBMS based on navigational model.

What happened in the 60's? In general, the business demands for some analytics style methods - companies need to make analysis from sales and transactions.
Doing this on navigational models was really too hard - also, we had DBMS but there was not a very good separation between logical and physical representation of the data. Low level representation of the database mirrored the logical representation, which is not necessarily a good idea - if you changed the physical organization you had to review the logical one too.
Not only that, existing databases lacked a theoretical foundation behind their organization - theoretical rules make a database strong.

What happened then? Edgar Codd was the father of relational algebra. He published the paper "A Relational Model of Data for Large Shared Data Banks, 1970 " where he introduced

the concept of tuples (an unordered set of attribute values - where in a database system a tuple is a row and an attribute is a column) and relations (mainly a synonym of tables, officially a collection of distinct tuples).
There were other elements as well:
Constraints are limits on the data that enforce consistency of the database - Key constraints are used to identify tuples and relationships between tuples.
Operations on relations (such as joins, projections, unions). These operations always return relations.
In practice, this means that a query on a table returns data in a tabular format.

Everything was organized and managed using a tabular format. Continuing on relational theory, normal forms were another key element.
The third normal form: non-key attributes must be dependent on “the key, the whole key, and nothing but the key”.

Transactions: the definition is a concurrent data change request on a database system.

Jim Gray definition: “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). ”

When we want to change something in the database, we want the change to be consistent and we do not want to change the idea of the data that is stored.
So, one of the main issues of transaction is to ensure consistency and integrity of the data that needs change.
This happens mostly in the relational databases that appeared around the Seventies of the 20th Century.
The solution is ACID transactions.
An ACID transaction should be:
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 in-progress transactions.
Durable: Once a transaction is saved to the database, its changes are expected to persist even if there is a failure of the operating system or hardware.

Example of an Acid Transaction:

I want to transfer 100 euros from X to Y.
The starting point is that X has 100 and Y 200.

If i want to move 100 from X to Y there are 2 tasks.
The first is to read, decrease and write X.
Then i have to read, update and write Y.

If the first task is successful and the second one fails, or vice versa, the result of the database is inconsistent.

Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

Now, we enter into the relational DBMS era. In this period SQL appeared, starting from IBM in its System R in 1974.
Then, there were a sort of Database Wars, mainly between IBM and ORACLE, for hegemony between systems.
By the mid eighties, most of the relational DBMSs adopted SQL as a standard query language because programmers were happy with it, it was easy to move applications and share queries.

Database Management Systems had a very important rule in the client server computing paradigm that started at the beginning of the 90s. The situation was that there was a server on the back end, with data and some logic, and then on the client (computers, mainframes, laptops) we had the program with the GUI. This was the period of Windows - windows started with the operative system, Bill Gates had the idea from Steve Jobs. Front end the systems were based on Windows.

Relational DBMSs in the Client-Server Computing ERA (late 1980s)

In the client-server model, presentation logic was hosted on a PC terminal typically running an operating system with a GUI.
Database systems 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.

On the client side we had the presentation logic, and the servers had databases. What happened was that if on the client side the applications were developed using different languages, on the backend the main language used for querying the data was SQL. SQL had a predominant rule, and the web was being born just in that period.

In that period there was a problem - that actually still exists: frustration of object oriented programming. If we use OOP we must consider some important features, encapsulation (we

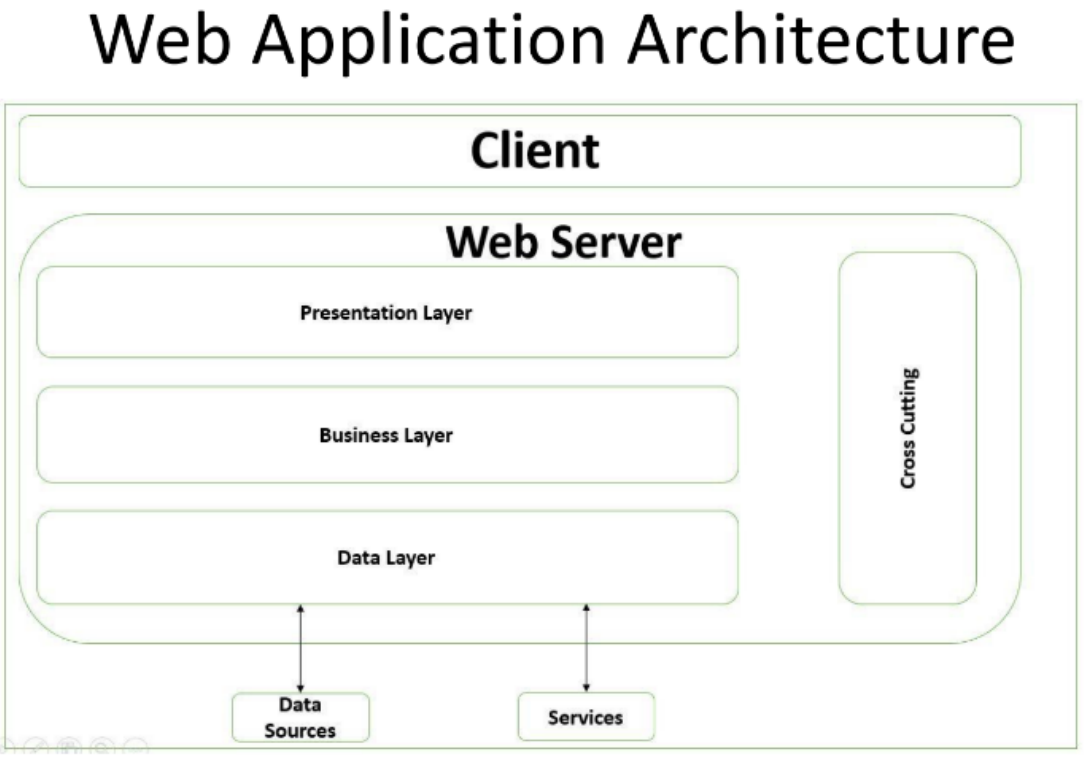
encapsulate objects into classes with public and private fields and methods) and inheritance (possibility of generating new objects by exploiting the hierarchy).
Object-oriented (OO) developers were frustrated by the mismatch between the object-oriented representations of their data within their programs and the relational representation within the database.
Whenever we must store an object into a relational database we often find ourselves having to break down the classes into small pieces to put every different piece in a different table in the DB.
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”
What were the solutions? There appeared an Object Oriented DBMS - OODBMS, which allowed to store program objects without normalization, and to load and store objects very easily. The implementation was similar to the navigation models but the system never gained market share and lost the war completely against RDBMS.
One solution is Object-Relational Mapping (ORM) frameworks (Hibernate for example), that alleviated the pain of OO programmers. using this tool we can define classes and work on them with set and get, and there are some methods that wrap around the queries. The tables are created accordingly.

Some more considerations.
From 1995 to 2005, no significant new databases were introduced, no new architecture for databases was introduced nor emerged - even though the internet started to pervade our life.
After 2005, social emdia and the era of massive web scale application started, and relational database models were not enough to support the requirements of new systems.

In the picture down here there is the main architecture of a web application. Massive web scale applications mainly involve big companies which need some requirements such as:

- Large volume of read and write operation
- Low latency response time
- High availability

Limits of RDBMSs for MWSAs:
if we want to improve the performance of a RDBMS system, we scale vertically by upgrading cpu and memory. This is costly and there are limits on the number of CPUs and memory dimension supported by a server. Even by doing this, it’s difficult to change the dimensions and scheme of the data.
What if we employ RDBMSs on multiple servers? It’s difficult to manage… what if i need to join two tables on two different servers? There are many performance issues. Finally, procedures for ACID transactions are hard to implement on multiple servers.



This puts down the basis for the third database revolution.
MWSAs serve a huge number of users and need large scalable DBMSs characterized by scalability, flexibility, availability, and low cost.

Why scalability? If we have a spike of traffic on a website we should try to distribute the load by adding a new server and shut it down when traffic becomes normal. It's costly sometimes!

Why flexibility? For example, if an e-commerce application handles several kinds of items, adding, removing or modifying one or more fields in the database is not an easy task because it might require modifying the scheme of the RDBMs and modifying the application code.

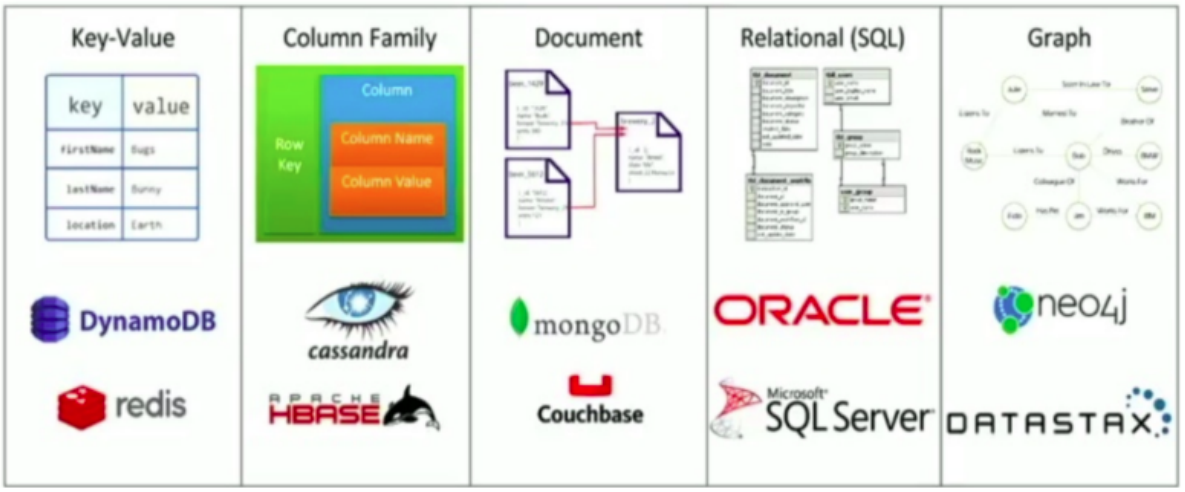
So, from the Scheme model, we move to Scheme less models.

Availability: if we consider an e-commerce, it must always be active or clients will not use it and we lose money. To manage this, we should deploy the whole system 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. The issue is that usually RDBMSs run on a single service with backups. This solution is not efficient because the second server does not improve the performance during the normal workflow.

Replicating and Sharding - dividing data in blocks and replicating blocks in different servers - is different than using a backup server, because it can distribute the load even during normal days.

In terms of costs, Most of the RDBMSs used by enterprises and organizations often have licensing costs. These costs depend on the number of users.
Using open source DBMSs could be a solution - you have to pay still for some services, but the costs are reduced.
Issues: sometimes this would require changing the whole architecture and changing the applications.

NoSQL is the acronym of Not Only SQL. It considers a set of data models and related software.
Most of NoSQL solutions ensures Scalability, Availability, Flexibility and are often open source.
ACID transactions may be not supported by NoSQL databases, but only by some of them
Roughly speaking, NoSQL databases reject the constraints of the relational model, including strict consistency and schemas.
There exists also NewSQL Databases which retain many features of the relational model but amend the underlying technology in significant ways.



So, three waves:
In the first one, in the 60s, we had the advent of the hierarchical database, network databases and ISAM files on mainframes and minicomputers.
In the second one the relational database was born, and new technologies and paradigms included client server architecture and the web 1.0
The third wave is seeing the rise of NoSQL, NewSQL, Big data platforms, and relational databases are sometimes still used - all together with cloud, social, big data, mobile, IOT.

For Nostalgics of SQL

Currently, scalable services for handling relational databases are offered by big enterprises such as Google and Amazon.
These services are cloud-based and pay-per-use services.
Using these services, enterprises may continue to use their software without the needs to migrate towards NoSQL databases.

The offered solutions are often scalable up to thousands of nodes and ensure high availability of the service.
Issues of SQL on Cloud Services
Even though the availability and scalability requirements are ensured, the flexibility is reduced as in classical SQL-based DBMSs.
Data and services are managed on the cloud, thus some privacy and security issues may arise.
Solutions with DBMSs installed and managed on site cannot be deployed.
Even though the costs seem to be reduced, a spending manager is needed for taking the costs under control.

ACID vs BASE

In general, the tasks of any kind of DBMS - and a database, of course? We need to store and retrieve data, but in order for things to work correctly the data must be stored persistently, it must be maintained in a consistent state (acid transactions!!) and of course in a correct and integer way - also, we have to ensure data availability.

Most of us experienced the usage of a DBMS on a single server. Most of the recent NoSQL DBMSs can be deployed and used on distributed systems, namely on multiple servers rather than a single machine. A distributed system is also called a cluster.
In order to exploit all the features - like making replicas and shards of the data - we will deploy systems on a distributed system.

Why would we like to exploit distributed systems?
These systems allow us to accomplish the requirements that led to the third database revolution. That is to say, they ensure scalability, flexibility, cost control and availability.
Another thing: It is easier to add or to remove nodes (horizontal scalability) rather than to add memory or to upgrade the CPUs of a single server (vertical scalability)
If we have the distributed approach, we have the possibility to implement fault tolerance strategies (if one server goes down, we can ask another one to do its job as well).

There are some problems, though.
First of all, if we want to ensure availability, flexibility, low latency… we must replicate data and create shards. But if i make a write on a server, it must be propagated between servers! If i have to update other copies, i should wait. So, if i wait, i lose in terms of performance and i will have higher latency. There is a tradeoff to be found between these things.
Another thing: if i have a cluster in a network, what if the network fails? I may have a simple disconnection of one server, or a disconnection between two sides. What can we do?
We will have to take in mind these problems and propose solutions to manage them.
We want distributed databases, we might have advantages but we must consider some issues to take under control.

Data persistence: we must be sure that once we have written something in a database, we can retrieve it even in case of failure or switched off application, cluster or server. This persistency can be achieved in memory, on the disk, on an ssd, on a tape…

What about data consistency? Let’s consider a classical example of a transaction. The status of the database should be correct before and after the modification. The transaction is either complete or not done.

Data availability: Data stored in a single server DBMS may be not available for several reasons, such as failures of the operating system, voltage drops, disks break down. In the figure we show a possible solution that ensures a high data availability: the two-phase commit.

1. Write to primary server
2. Copy to backup server
3. Acknowledge copy complete
4. Finish write operation

If the primary server goes down, the Backup server takes its place and the DBMS continues to offer its services to the users.
This takes more time but we need it if we want more consistency and availability. We lose in terms of performance. We cannot have everything.

The two phase commit is a transaction that takes more than double the time of a transaction on one server. This is because of network overhead.
You can have consistent data, and you can have a high-availability database, but transactions will require longer times to execute than if you did not have those requirements.

There are some applications in which it is not acceptable to wait too much time for having concurrently consistent data and highly available systems.
In this kind of application, the availability of the system and its fast response is more important than having consistent data on the different servers.
There are applications, like e-commerce and social networks, which need responsiveness. It is much more important to have low latency than instant consistency.

If our main interest is availability, we can use two servers - one primary, one secondary - and when I make a write, the first and second steps are to write in the primary server and return control to the customer after the write is finished. Then, some thread of the system will update the second server. We don't care about this. This is called eventual consistency - there might be a period of time where copies of data have different values and in case of severe malfunction some things might change, but eventually all copies will have the same value and the customer will not see their experience interrupted.

Available but not consistent:
In the e-commerce scenario the shopping cart may have a backup copy of the cart data that is out of sync with the primary copy.
The data would still be available if the primary server failed.
The data on the backup server would be inconsistent with data on the primary server if the primary server failed prior to updating the backup server.

One aspect of availability is that i cannot work with a server when it is working.
When dealing with two-phase commits we can have consistency but at the risk of the most recent data not being available for a brief period of time.

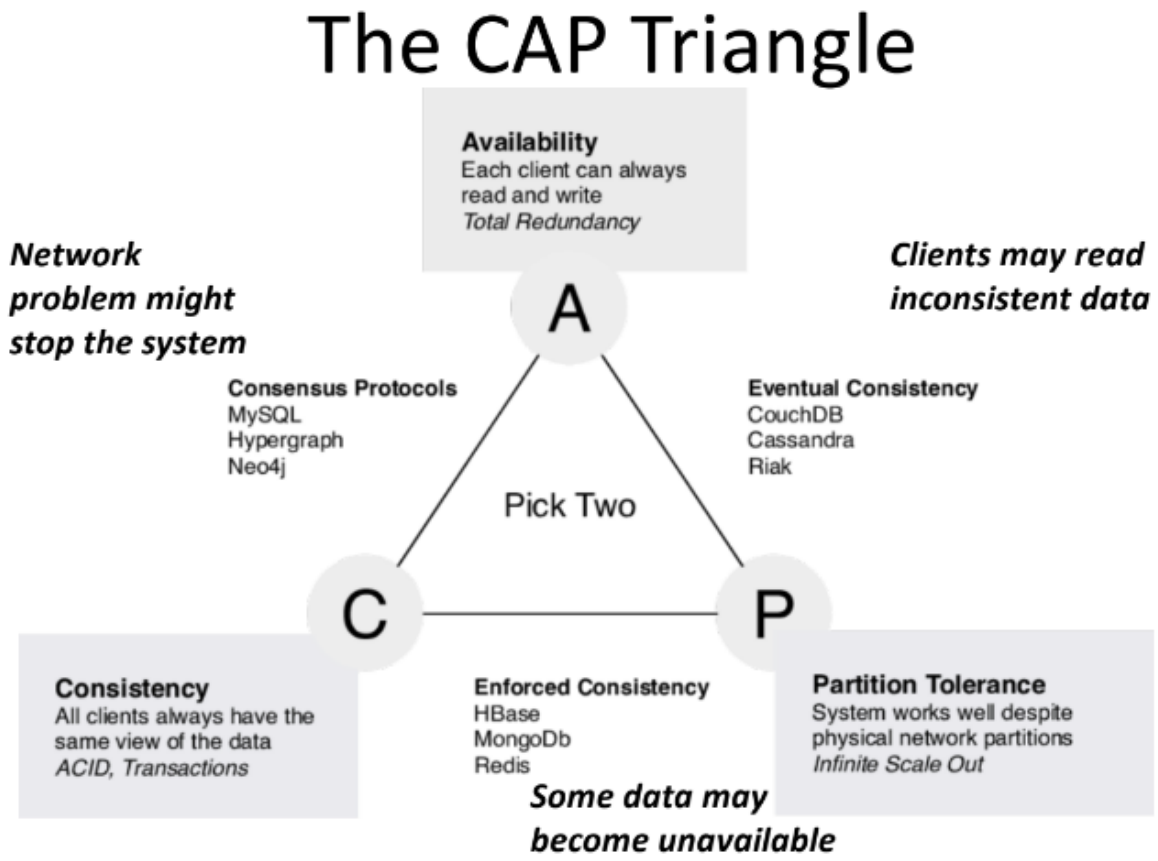
While the two-phase commit is executing, other queries to the data are blocked.
The updated data is unavailable until the two-phase commit finishes. This favors consistency over availability.

What if we have a cluster of servers? There is a situation where a network is partitioned. Let's suppose that we have a distributed server network divided in two parts with shards and everything, and two users, A and B. What do we want? Consistency or availability?
Consistency: we deactivate one of the two subnetworks while it is updated, and either A or B cannot access the service.
Availability: i have to accept that maybe user A and B will not see the same data for a while, but they will both access the service.

The CAP Theorem:
Distributed Databases - but in general a distributed system - cannot ensure at the same time:

- Consistency (C), the presence of consistent copies of data on different servers
- Availability (A), namely to providing a response to any query
- Partition protection (P), Failures of individual nodes or connections between nodes do not impact the system as a whole.

At maximum two of the previous features may be found in a distributed database.
This is the synthesis of everything we have discussed until now.



If we allow Availability and Partition tolerance, clients may read inconsistent data

If we allow Availability and Consistency network problems might stop the system
if we allow Consistency and Partition Tolerance, some data may become unavailable

If we consider Consistency and Availability as priorities, all nodes are always in contact, when a partition occurs, the system blocks.
We are in the world of ACID transactions and relational databases. use cases are Banking and Finance applications, systems which must have transactions e.g. connected to RDBMS.

If we consider Partition Tolerance and Availability to be the priorities, we sacrifice consistency.
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. Indeed, this system state will also accept writes that can be processed later when the partition is resolved.
A query could not return the most recent data. Use case: shopping carts.

If we consider Partition Tolerance and Consistency to be the priorities, we sacrifice availability.
As suitable for applications which require consistency, but also partition tolerance, while somewhat long response times are acceptable, like Bank ATMs. They are usually based on distributed and replicated relational or NoSQL systems.

This is one of the starting points of the non functional requirements when discussing a project. In most of the recent DBMSs there is a framework supporting NoSQL architectures, we will be able to set up a level of consistency and availability. In mongo we can set up the number of replicas. and we can set how and when to update the replicas.

Since on one side we had ACID properties, we will consider BASE properties, typical of AP systems.

- 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 data replication).
- S stands for soft state: data stored in the nodes may be updated with more recent data because of the eventual consistency model (no user writes may be responsible for the updating!!) so data might not be in a strong consistent state, but it will change.
- E stands for eventually consistent: at some point in the future, data in all nodes will converge to a consistent state.

From one side we have ACID transactions, not always right for large scale applications. We move towards BASE properties of systems.

The Latency issue:
If I want high availability, a requirement of modern applications, I need replicas. if i implement replicas i must take care about the consistency - the stronger the consistency the higher the latency. so, if I have to ensure strong consistency on all replicas, I will lose in performance. This is the reason why the level of consistency is usually tuneable. so, engineers of the system should take this aspect in mind. The higher the replicas, the higher availability, the harder to set up consistency level.

There are different types of eventual consistency.
The simplest one is Read-Your-Writes Consistency. so, we decided to implement eventual consistency. We do not expect to always have consistent data. Still, there are several shapes of eventual consistency. As an engineer, we must choose which type of consistency we want and then look up for that specific framework to ensure that kind of consistency. Remember that before implementing we must do a feasibility study in order to see whether something is actually implementable - also to ensure a certain budget.

so, **Read-Your-Writes** Consistency ensures that if a specific user updates a value through a write operation, all their reads of that record will return an updated value.

Example: Alice updates a customer's outstanding balance to 100 euros. The update is written to one server, the replication process updates the other copies during time. Every Time Alice queries the customer's balance, she will see 100 euros.

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. Of course, eventually the data will be updated.

Monotonic read consistency ensures that if a user makes a query and sees a certain result, all the users will never see an earlier version of the value. So, when a query is made, everything is updated.

Example: Alice updates a customer's outstanding balance to 200 euros. The update is written to one server, the replication process updates the other copies during time. Every Time Bob queries the customer's balance, he will see 200 euros even if the balance has not been updated in each server.

Monotonic write consistency ensures that if a user makes several update commands, they will be executed in the order they issued them. The update might take some time, but the order is ensured.

Example: Alice updates again the customer balance removing 10% of the balance. The customer balance, from 200, is now 180. Charlie, one of her customers, orders 100 euros of material. His outstanding balance goes from 200 to 180 and then to 280.

if the operations took place in another order, Charlie's balance would go from 200 to 300 and then 270.

If an operation **logically depends** on a preceding operation, there is a causal relationship between the operations. **Causal consistency** ensures that users will observe results that are consistent with the causal relationships.

The example is the order of comments under a picture of a social network - if they are totally mixed it does not make sense to follow the conversation.

NoSQL Databases Architectures

Key-value databases: introduction

This is one of the simplest architectures. It is a kind of architecture characterized by simplicity and flexibility. We will see that it can be used for very simple requirements.
The starting point to understand this database is recalling the concept of an array.

An array is an ordered list of values. Each value in the array is associated with an integer index. The values are all the same type
Limitations:
The index can only be an integer. The values must all have the same type.

From an array we can go to the associative array, which is a collection of key + value elements, where keys have the same role as indexes in arrays but can be generic identifiers such as strings or integers, and values can be of different types.

Key Value Databases are based on associative arrays, but stored in a persistent way on a disk. When we have this kind of architecture we do not have to define tables or schemes. Maybe we are working with data that does not need to be grouped considering values and attributes.
What can we do? We can retrieve a value associated with a key - of course, each value must have a unique identifier in the form of the key.
The DBMS, or the java library which allows us to work with this database and is mainly based on files, which represent the persistent storage, then a java API to make the queries, and then the fact that the database is most of the time in memory.
So, given a key we associate it to one and only one value, and vice versa.
The role of the key is the same than in relational databases.
Usually, when we work with these databases, we define a namespace - we define a “bucket”, which has the same role as a table, but it is not actually a table, more like a collection of keys and values. In each bucket, so in each namespace, all keys are unique. Keys can repeat in different namespaces/buckets.
The design of a key value database can be summed up in being able to design keys in a correct way.

The essential features of Key value databases are Simplicity, Speed, Scalability.
Developers tend 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.

We will analyze the features one by one, and we know that we use these databases when we are mainly interested in speed of access and simplicity of the data setting, not complexity of the data structure.
Let’s suppose we have a word processor like Microsoft Word or open office, that gives several features to the user in order to format the text in several ways. If i need to write a thesis or an essay we need those features, but if we only need to write down a shopping list all those options are a bit too much.

Simplicity is one of the main features for when we need a schemaless approach. Key value databases allow us to modify the code of our program without modifying the database. We can have an application where the database is composed by several parts depending on the different needs of the data. For example, the configuration file setting of an application can be stored and obtained with this.

If we do not need all the f**eatures of the relational models** (joining tables or running queries about multiple entities in the database), we may exploit key-value databases
We do not have to **define** a database **schema** nor to define data **types** for each **attribute**.
We can simply modify the code of our program if we need to handle **new attributes, without the necessity of modifying the database**.
Key-value databases are **flexible** and **forgiving**: we can make mistakes assigning a wrong type of data to an attribute or use different type of data for a specific attribute.

Speed is also very important, for when we experience peaks of data access, which with big data can increase the latency. To reduce it, the best way is to have data ready in memory. So, most of the DBMS software based on key value databases maintains data in memory.
The idea is that when we start the application from the disk, the database is loaded in memory - or the application starts and the cache memory, while we are working with the application, will store the data, which will eventually be updated in the disk as well.
So, if we make a read operation first we read in memory and eventually we check in the disk and fetch the data.
For the write operation, first we write on the memory and then the control returns to the application. Meanwhile the changes are sent to the disk to maintain consistency.
In this case we ensure a lower latency.

Obviously, the RAM is not infinite, so while the application is working we cannot fill the RAM indefinitely. Something will have to be removed, but what? There are several algorithms and strategies to manage replacing data in the cache, and the situation here is similar. The least recently used (LRU) is one of the most used algorithms: data that has not been used recently will be removed.
For example, if we take as an example a shopping cart application, an untouched cart will eventually be removed from the **cache** of the server.

Obviously, this kind of database can be distributed, and easily scalable. What does it mean: we can easily add and remove servers to accommodate the load of the system. There are several strategies to add and remove servers without any particular problems on the application's side. One important required thing: analyzing the amount of read and write operation - first estimate and then check. Depending on read and write numbers we might decide on architecture and servers number.

scalability is the capability to add or remove servers from a cluster of servers as needed to accommodate the load on the system.
When dealing with NoSQL databases, employed in web and other large-scale applications, it is particularly important to correctly handle both reads and writes when the system has to be scaled.
In particular, in the framework of key-value database, two main approaches are adopted, namely Master-Slave Replication and Masterless Replication.

If we consider that key value databases are very large associative arrays, we can imagine that it is not difficult to make shards and replicas - whilst making shards of relational tables is difficult because of the join feature.
An interesting thing with columnar and key value databases: you can replicate a relational database without all the controls on integrity, consistency, type. It is a very wrong approach though.

Master Slave replication: in this architecture we have a master server in charge to accept write and read requests. The slaves only accept read operations.
Whenever the master receives a write operation request, it will propagate the request to the server with the replicas in order to update them - using the concept of eventual consistency.
So, this architecture works well for applications with higher numbers of read operations. This is because we can add a lot of servers without conflicts, since consistency is managed by the master alone. Write operations do not have to be coordinated.

Pros:

- Very simple architecture: only the master communicates with other servers.
- There is no need to coordinate write operations or resolve conflicts between multiple servers accepting writes.

Cons:

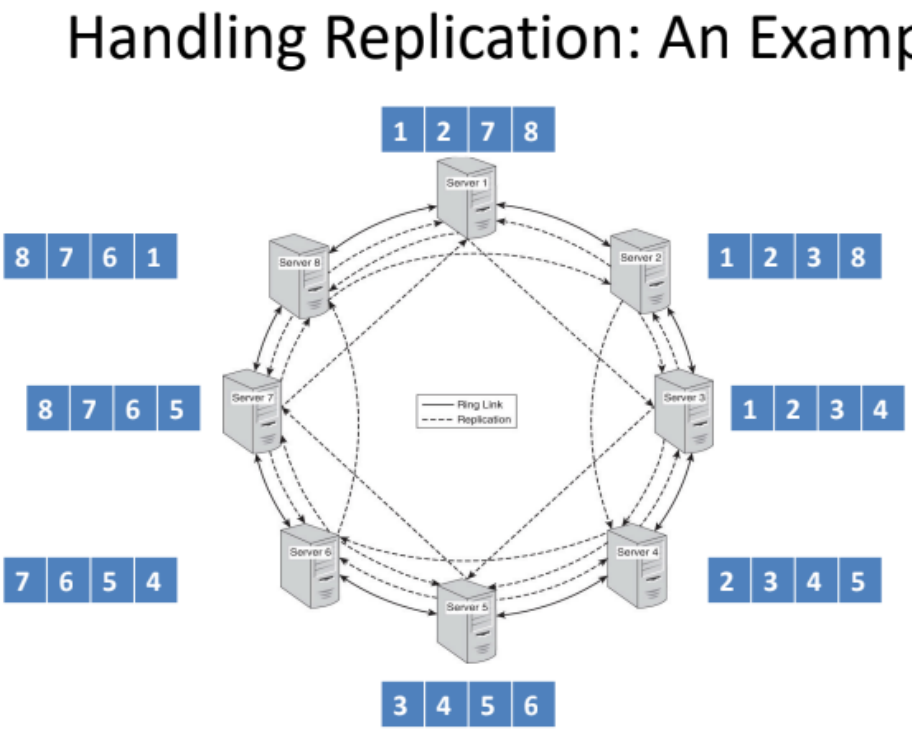
- if the master fails, the cluster cannot accept writes.
- In this case, the entire system fails or at least loses a critical capacity, such as accepting writes.

So, we might add some complexity by allowing for the slaves to communicate among them.
Each single slave server may send a simple message to ask a random server in the cluster if it is still active, and to ask each other if they have heard from the master. If they realize that there are no news from the master, they can start a protocol to promote a new slave to the status of master, to ensure functional requirements - there might be problems in handling the load but better than nothing!!

After this, we will see the architecture called Masterless Replication and can be applied to any kind of DBMS. For example, Mongo was based on a master-slave architecture until recently.
This can be considered a general aspect of databases.
If we experience a higher number of writes we could have peaks towards the master. In that case we could have problems in ensuring a good latency, whilst the slaves do not have that much traffic but cannot help with write operations. Imagine a system like Ticketmaster - or even Agenda Didattica.
We consider an architecture in which all nodes in the cluster accept read and write operations. We can have a ring-like logical view of the architecture, where any server can communicate with the server on their side - this is not compulsory of course. The point is that there is no master, no server that has the master copy of the data. Also, not all data is replicated on all servers, we have both replicas and shards. The main idea is that each node has their neighbors.
Let's make an example:
Each time there is a write operation to one of the servers, it replicates that change to the three other servers holding its replica (we consider 4 data replicas).

Each server replicates to its two neighbors and to the server two links ahead.

The important thing is to remind that each server may respond to a write operation, then we will see that for example, each server may answer for a specific ticket (the same type of query for example).



We suppose to have our dataset divided into 8 chunks. We have 8 servers and each one of them hosts 4 replicas. We want to exploit the philosophy introduced by masterless replication as in, each server updates his two direct neighbors and the ones after one step.
So, if we stay in server 1, and we write on partition 1, server 1 will update the value on server 8, 2, 3 - which could still be updated by server 2. If for any reason the write operation is answered by server 2, he updates 1 and 2, and server 1 updates 8. There are several policies we can implement.

Now we see some guidelines on how to build keys, which, as we already said, are the most important part of designing a functional key-value database.
The key to the associative array has the same role as the primary key in a relational database. In the framework of key value databases, adopting meaningful keys is more preferable than utilizing meaningless primary keys (which is a good practice in relational databases, since keys should stay immutable there).
This is because in key value databases there are no tables, no columns, no predefined attributes; only keys and values. A specific value is only associated with its key and can only be retrieved thanks to it.
Since in key value databases we are using namespaces, we may define more interpretable namespaces that give us more information. We can build keys that embed information regarding

- the name of the entity
- the entity identifier
- the attributes of the entity.

The general rule for defining keys is the following (single entity, no relationships):
EntityName : EntityId : EntityAttribute

For each key, we may store in the database a specific value.
For handling customer information, we can use just one namespace (table).
We may also exploit replicas and sharding.

Power of being schemaless: I can choose which attribute I want to use. In the same namespace I can also put other entities. We can also build relationships.

Question: the data could be also a json object or other complex data - can we say that the key stays similar whilst the value is a json with all the info? If we are working with Json we can just use a document database. if we say that the json always has a certain structure we are defining a scheme… it might be better to use another type of database. on the disk, all this info will be very close. so if we must retrieve just a bit of the info, we must retrieve the whole json.

Now, let's move at a lower level for a while. If I have a key - that can be anything but in general is a string - then, the system has to elaborate it in order to extract the value - that is, to identify the address in memory, or on the disk, of the value. So the string must be elaborated. In a simple array there is an index and it's easier!
IN KVD we use hash functions to obtain an address. The hash function returns a number from an arbitrary key - an integer that can be used to locate the data into the memory.
Of course hash functions have some limitations - values returned by hash functions only look like random values and could not be unique - this can lead to collision problems!

An example of hash mapping can have couples key + hash value (a number in hexadecimal format).
We can also use hashing to select servers:
We would like to balance the write loads among the servers in a masterless replication architecture.
For the sake of simplicity, let consider the ID of a specific server as the address to identify.
If we divide the Hash Value by the number of servers, we can collect the reminder, namely the modulus.
The modulo span from 0 to 7. Each of the 8 servers can be assigned a number (the address) between 1 and 8.

In this way, the system chooses the server that answers to the query, and the load is more distributed. We can see an application: if we sell tickets to events, we want to avoid two people buying the same seat. So, we take the key for the ticket - anyone trying to purchase the seat on the same day generates the same key, which gets transformed by the hash function and modulo function in the same server address. So, the same seat always gets processed by the same server on the same day and there are no collisions!

Key-value databases do not expect us to specify types for the values we want to store.
If we want to specify an address for a customer, for example, we can use an attribute for each piece of the address, or separate the town from the street… we can use the format that we prefer.

Let's conclude with some considerations.
First of all, we must consider the recurrence of our application. Once we decide that we can use a key value database, then we give a look and make a feasibility study regarding the options and toolbox of the frameworks that we find. There might be some limits on the different DBMSs and frameworks, regarding length of the keys, dimension of the values, possibility to make ACID transactions…
So, we must read the official documentation of the tools.
Among different key value databases, there might be different features that we might need or not - we must always consider the tradeoffs between various systems and features.
Another limit, of course, might be the price itself of the tool.
Our application requirements should be considered when weighing the advantages and disadvantages of different database systems.

Which operations are allowed directly in key value databases?
Mainly retrieval, setting and deletion of a value by key. If we want to do more - such as search for a person which has the name "Paolo" - we have to do that with an application program. Remember that with the NoSQL databases we are moving back towards denormalization, working more at the application side, not having theories behind databases, and not having constraints.

Complex Queries with KV Database

```
appData[cust:9877:address] = '1232 NE River Ave, St.
  Louis, MO'
```

```
define findCustomerWithCity(p_startID, p_endID, p_City):
    begin
    # first, create an empty list variable to hold all
    # addresses that match the city name
    returnList = ();
    # loop through a range of identifiers and build keys
    # to look up address values then test each address
    # using the inString function to see if the city name
    # passed in the p_City parameter is in the address
    # string. If it is, add it to the list of  addresses
    # to return
    for id in p_startID to p_endID:
         address = appData['cust:' + id + ':address'];
         if inString(p_City, Address):
            addToList(Address,returnList );
    # after checking all addresses in the ranges specified
    # by the start and end ID return the list of addresses
    # with the specified city name.
    return(returnList);
end;
```

This is an example of a complex query. We obtain a value and have a function in pseudocode that retrieves all the addresses in St Louis.
We define an empty list and, in a loop, searching through IDs, we retrieve addresses and build the key. We search for the city in the string address. It is time consuming to do this.
I wrote a new code, with the possibility of making errors.
If I realize that I'm writing too much code, I chose the wrong database.

There are some DBMSs which are key value databases but exploit the concept of indexes. An index is a data structure that allows us to quickly retrieve data.
For example, a built-in search system could index string values stored inside the database to create an index for rapid retrieval, by keeping a list of words with the keys of each key value pair in which that word appears. Of course, this structure must be maintained and must be updated with new inserts.
If I have more readings than writings, indexes are convenient; the more writings I have, the more the performance is impacted.

Key-value databases: insights

Some additional look inside the architecture, mainly how to work with partitions and replicas, and hashing strategies.
Everything is based on a key in which we specify a lot of information.
For example, today we will see how to codify one to many relationships inside keys. Why one to many? Because if there are many much more complex relationships maybe this database is not the best one to use.
Values can be almost anything, it depends on the application and on how we would like to use the data.
Hashing functions to generate the address of the data are very important and can be also used to distribute the loads among different servers.
We keep in mind that values returned by hash functions may not be unique and generate collision problems.

What about keys? They are usually strings, long strings - but they should not be too long. Long keys will use a lot of memory (and key value databases are very memory intensive already), but short keys are more likely to lead to conflicts in key names and in hash functions.
Remember also that keys must be meaningful.

What about values? Values are objects - at a low level, a collection of bytes - linked to a key. They can be complex, but we must be careful with those since they will be stored in memory and disk, and retrieving them can be labor intensive. It all ultimately depends on the application; there might even be limits on the dimension of a single value, fixed by the frameworks for KVD.

What is a namespace? 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 enable duplicate keys to exist without causing conflicts by maintaining separate collections of keys.

They are also helpful when multiple applications use the same key-value database, and allow us to organize data into subunits.
if the same application has two modules - for example, one tracks the orders of the users and the other one manages the specific order. In the two different modules we can have different values for the same key (for example, in the first module we can have “clothing” as the value, and in the second we can have “levi jeans”).
These two modules can have two different groups of developers. Of course at a low level each key has the namespace as a prefix.
This is not like using two tables on a relational database! Here the workflow is split into two different separate things.

Concept of data chunking - or data partitioning. Given a dataset, we divide it into pieces and we deploy data on different servers. As a first example, we pretend to have two servers. We divide the data in two by the value of the key put in lexicographic order.
Of course, a cluster may contain more than one partition, and several strategies exist.
The main objective to partition data is to balance read and write operation among servers - or in general, the workload.

Partition keys are keys who identify the specific partition in which the value has been stored. In a key value database, any key is used as a partition key - for example, a set of attributes or a part of an attribute. Usually, hashing is a good strategy for identifying the partition on which the data is stored. (Example of the tickets).

Another important element to discuss is flexibility - key value databases are flexible. By modeling the key we can decide how to use the architecture to specify the attribute of an object.
We are not required to define all the keys and types of values we will use prior to adding them to the database; we might decide to change how to store the attributes of a specific entity. We might decide to add attributes starting from one point on forward.
From a data standpoint, we can change the way of handling data and attributes whenever we want; from an application standpoint, we need to update the code in order to handle data differently.
Doing this with a relational database would be much more difficult. The counterpart is that in this case handling application code is more difficult.

We want to put our namespaces, our buckets, on a cluster. Usually there are some architectures which are master slave based, others that work on masterless clusters.
The important thing is that usually the servers of the clusters are loosely coupled: this means that they are fairly independent and just exchange some messages with other clusters, they require minimal coordination with other servers.
Each server is responsible for the operations on its own partitions and routinely sends messages to each other to indicate they are still functioning.
The data should be designed to limit the interaction between servers whenever possible. The important thing is that when a node fails the others can take over its activity.

If we, in a masterless architecture, exploit the logical ring organization of the servers, we can ensure availability. If we mate a write operation by selecting a server, using the eventual

consistency approach the server will propagate the write operation to the adjacent ones. If the server fails, both adjacent servers can respond to read and write requests for the data on the unavailable one. When the original server comes back online, it will be updated by the ones that substituted it.
So, this strategy of replication exploiting the hashing function ensures on one side availability and load balancing.

Obviously, if we work with replicas, we know that high availability depends on the number of replicas that we use.
If the number of replicas is high, the probability of losing data is low but the system will perform less in terms of response time.
On the other hand, if the number of replicas is lower, the probability of losing data is higher but the system will perform more in terms of response time. Having a lower number of replicas might not be a problem if data is regenerated and reloaded easily.
In conclusion, depending on the requirements the number of replicas is a parameter to set.
The important thing for us is to have a snapshot of what might happen.

Most of the frameworks that exploit NoSQL Dbs give us the opportunity to select:

- the number N of replicas that we want to use
- the number W of write updates that must be done before returning the control to the application
- the number R of copies you need to read before assuming that your reading is correct.

For example, if we have N=3 we can have different settings.

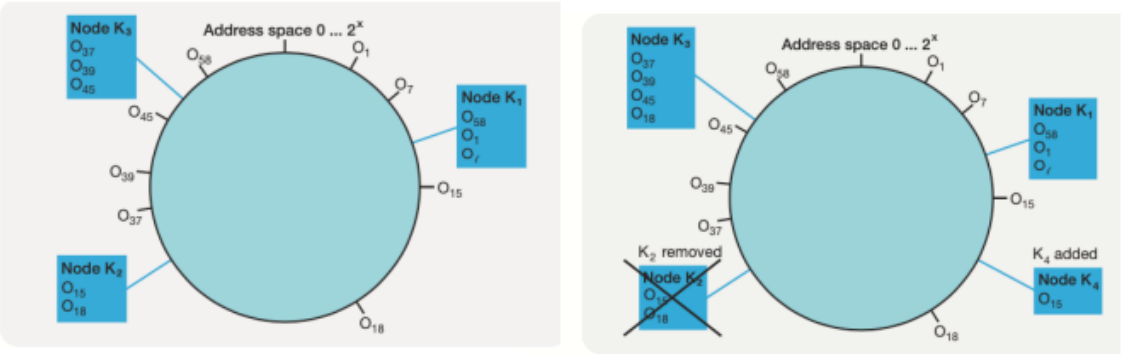
- N = 3, W = 3, R = 1. Before returning control to the application we update all the copies. Slow writes, fast reads, consistency;
- N = 3, W = 2, R = 2. We maintain a good level of consistency and the writes are faster. Reads are slower. if we read just once we are not sure that we get the updated value, if we read twice we are sure.
- N = 3, W = 1, R = 3. Write operations are the fastest, reads are slow because we read all copies to make sure we have the latest version. Data might be lost if a node fails before the second write. Reads are slow but consistent.
- N = 3, W = 1, R = 1. Highest availability, lowest latency, problems with consistency.

What do we choose? It depends.

Hashing functions are important, in distributed databases, to manage shards and replicas.
Given a key, there will be a low level procedure that transforms the key into a hash value applying a function.
Let’s remember that small changes in the key give large changes in the hash value - this is because hash functions are designed to distribute inputs evenly over the set of all possible outputs - and the output space can be quite large.
For example, it can happen that two similar or related pieces of information end up on different servers because of the difference in hash function.
How do we resolve the collision problem (we may have different values with the same hash function value because 2 different keys might give the same hash value): we get a list in

which we specify key and value corresponding to each hashed key. This adds some complexity, because if i want to write i might have to add a list element.
Of course, if we have a large scale dataset and we use moderate length strings for keys, collisions are less probable.

Another problem that we have is: let's suppose that we have 8 servers and we distribute the load by doing the module on the hash value. What if we add or remove a server, even for a short time period? We need to change the hashing function, or at least the module, and relocate all the records of the database.
We need a new setup period to relocate everything, and that means stopping the system.
Another solution is to exploit consistent hashing, which is to remap only the keys mapped to the neighbors of the new node.
So, if we have 4 nodes and we want to add another one between 4 and 1, we relocate some of the keys that were in 4 and some of the keys that were in 1 without touching nodes 2 and 3. Of course, the more you scale it the less impact it has on the system. Imagine having hundreds of servers!
Also, we do not need to change the hashing functions.
The key idea is that we calculate the hashing value, so we identify a position in the circular array containing the keys and the IDs of the servers.
We must have the range of the hash values, which has to be the same for all values, and we get the distribution of the keys. Then we calculate the position into the array of the id of the server, so that each server has a position in the ring. A specific server hosts all the keys which precede it - the server associated with a specific key is its successor in the hashing/address space.



The principle is to calculate the hashing function of the name of the server as if it were a key.

If we ask: "How and why do we use hashing?" the answer might be "We use hashing to balance the load on the servers". If we add or remove a server, we can use Consistent Hashing.

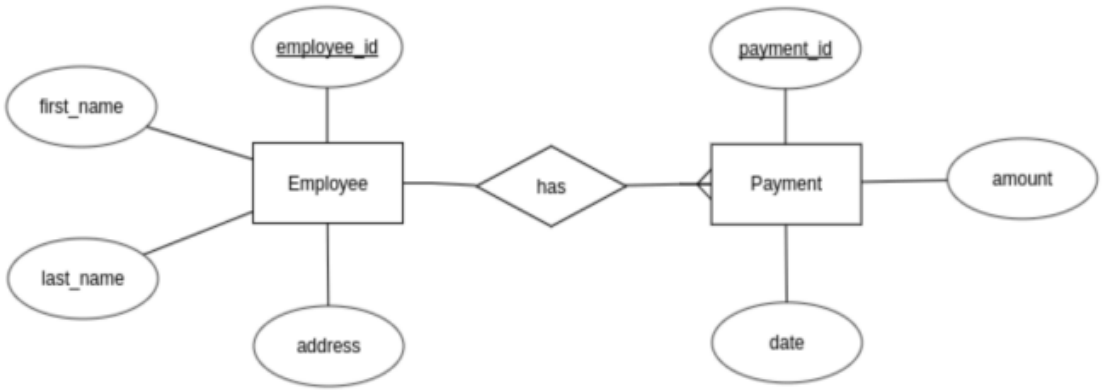
We say that key value databases are mostly used because they can be exploited as a cache, for retrieving data from the main memory.
Operating systems can exploit virtual memory management, but that entails writing data to disk or flash storage. Reading from and writing to disk is significantly slower than reading from RAM memory.
But memory is not infinite - ok we can use strategies to free it, but when it is not enough we should use data compression.

We must consider a tradeoff with compression level and execution time for compressing data. The more the data is compressed, the more time it takes to compress it.

Some considerations.
If we need data organization and rules in the management of the data, relational databases are a more suitable choice.
If we want high availability (ensured by replicas), low latency (ensured by the memory part), simplicity and flexibility, key value databases, even with other databases, may be used. So simplicity is the key word in this case.

Now, an example on how we can handle one to many relationships with key value databases.

Let consider the following data structures:



We have the Employee entity, and the entity Payment. One employee can have one or more payments.
In relational databases, we can use two tables with a foreign key in the Payment table - which is the Employee id. We should query with a time-consuming join.
Now, we want to translate the data modeling from a relational model to a key-value model.
Take in mind that keys embed information regarding Entity Name, Entity Identifier and Entity Attributes.

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

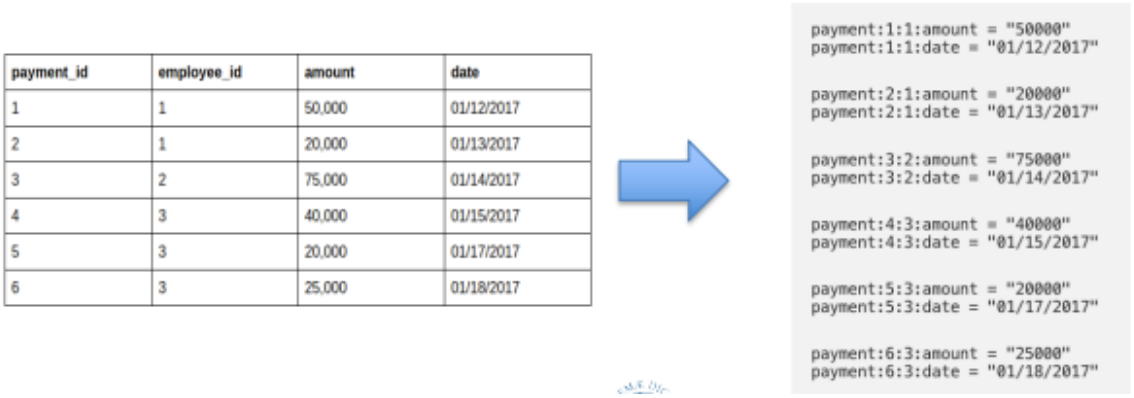
```
employee:$employee_id:$attribute_name = $value

employee:1:first_name = "John"
employee:1:last_name = "Doe"
employee:1:address = "New York"

employee:2:first_name = "Benjamin"
employee:2:last_name = "Button"
employee:2:address = "Chicago"

employee:3:first_name = "Mycroft"
employee:3:last_name = "Holmes"
employee:3:address = "London"
```

In the same namespace we can create the collection of key values for the one to many relationship by setting the following key - value configuration:
payment:$payment_id:$employee_id:$attribute_name = $value



payment_id	employee_id	amount	date
1	1	50,000	01/12/2017
2	1	20,000	01/13/2017
3	2	75,000	01/14/2017
4	3	40,000	01/15/2017
5	3	20,000	01/17/2017
6	3	25,000	01/18/2017

```
payment:1:1:amount = "50000"
payment:1:1:date = "01/12/2017"

payment:2:1:amount = "20000"
payment:2:1:date = "01/13/2017"

payment:3:2:amount = "75000"
payment:3:2:date = "01/14/2017"

payment:4:3:amount = "40000"
payment:4:3:date = "01/15/2017"

payment:5:3:amount = "20000"
payment:5:3:date = "01/17/2017"

payment:6:3:amount = "25000"
payment:6:3:date = "01/18/2017"
```

At the end of the translation process, data will be organized in a unique bucket.
Basically, we have put the id of the employee inside the key of the payment.
Remember that more complications on the querying of this kind of database has to be handled by the programmer by scanning the bucket and searching for values in the keys. If we are programming too much, then maybe we chose the wrong database.
Remember: it is an associative array, we lose the order, everything is handled by the key, we need to know the key of the records.
All structures that have to be translated into key value databases are very simple. One to many relationships can be tolerated, not much more.

Key-value databases: design tips and case study

It is very important to pay special attention to the keys, which represent the data modeling. If we insert too many things and write too much code, maybe this kind of database is not the right one.

There are some naming conventions forkeys, namely: the naming components of the keys should be meaningful and unambiguous. When we would like to retrieve ranges of values, range based components should always be used.
Also, when appending components to make a key, the delimiter should always be the same in every occasion.
Whilst maintaining these rules valid, try to keep the keys as short as possible - the longer keys are, the more memory they fill.

Well-designed key pattern helps minimize the amount of code a developer needs to write to create functions that access and set values.

Another suggestion is to define set and get functions to use them in higher level modules, to improve the readability of the code and to reduce the repeated use of low level operations.

It is very important to check and handle errors into the code. The software must work and be tested. Especially in production applications.

```
define getCustAttr(p_id, p_attrName)
    v_key = 'cust' + ':' + p_id + ':' + p_attrName;
    return(AppNameSpace[v_key]);
```

```
define setCustAttr(p_id, p_attrName, p_value)
    v_key = 'cust' + ':' + p_id + ':' + p_attrName
    AppNameSpace[v_key] = p_value
```

AppNameSpace is the name of the ***namespace*** holding keys and values for this application.

Getting and setting values, as it can be seen here, are two similar operations. The get builds a key and searches for it, and returns it. The set takes a value as an input as well, then puts it in the database after building the key.
These two functions are a basis for the usage of this kind of database. Everything should be made by hand.
If we want to implement an incremental ID, we can still use these functions and only add these things. The end result is a database management system made by hand but very fast.

First thing to do: work with objects on the code, define set and get.

We may have the necessity to have an application with customers and purchases, and we would like to retrieve all the customers that made an order on a specific date.
Let's imagine that we want to exploit the key value db to retrieve all the customers that made those orders.

We can use the key describing the fact that a customer made an order by specifying, after the customer, the date, like " cust20220627:1:CustID ". This kind of key is useful to query ranges of keys, because writing a function that retrieves a range of values is easy.

Let's see the code, where the date is the input. We create the list that will contain the IDs, and a counter for the range.
Then we build the key.
After building the key we start searching into a loop, in which we append the retrieved value from the namespace into the list created in the beginning. Then we build another key and we begin again.

In the end we return the list.

Still, if it gets too complicated then the database we chose is the wrong one.

The following function retrieves a list of customerIDs who made purchases on a particular date:

```
define getCustPurchByDate(p_date)
    v_custList = makeEmptyList();
    v_rangeCnt = 1;

    v_key = 'cust:' + p_date + ':' + v_rangeCnt +
      ':custId';
    while exists(v_key)
        v_custList.append(myAppNS[v_key]);
        v_rangeCnt = v_rangeCnt + 1;
        v_key = 'cust:' + p_date + ':' + v_rangeCnt +
          ':custId';

    return(v_custList);
```

Let's talk for a second about simple or complex values. Let's suppose we want to retrieve the information about a customer. The customer is defined in the key.
Here is the code:

Consider the following function which retrieves both the name and the address:

```
define getCustNameAddr(p_id)
    v_fname = getCustAttr(p_id,'fname');
    v_lname = getCustAttr(p_id,'lname');
    v_addr  = getCustAttr(p_id,'addr');
    v_city = getCustAttr(p_id,'city');
    v_state = getCustAttr(p_id,'state');
    v_zip = getCustAttr(p_id,'zip');
    v_fullName = v_fname + ' ' + v_lname;
    v_fullAddr = v_city + ' ' + v_state + ' ' + v_zip;
    return(makeList(v_fullName, v_fullAddr);
```

The function makes 6 access to the database (on the disk) using the `getCustAttr` function.

To speed up this function we should use caching and reduce the number of times the developer calls getCustAttr.
If we always show together the full name and address, should we maintain these attributes separately? Maybe not. A good idea could be to store together all the commonly used attributes. Key value databases usually store the entire list together in a "data block" which allows us to access just 1 block rather than six.
So, when deciding which attributes to put together and which to separate, we should know which attributes are most commonly searched together.

If we use very complex values, we could risk that, for example with a json file, there are many things inside. It is a good approach for document databases. Using this solution in key value database is that this information could be stored on different pieces of the disk: even if we make a unique set or get call, we might need more accesses to the disk - this means more problems in terms of performance.

Limitations of key value databases:
The only way to look up values is by key
Some DBMSs for key-value DB offer APIs that support common search features, such as wildcard searches, proximity searches, range searches, and Boolean operators. Search functions return a set of keys that have associated values that satisfy the search criteria.
Some key-value databases do not support range queries.
Some DBMSs (ordered KV databases) keep a sorted structure that allows for range queries and/or support secondary indexes and some text search.
There is no standard query language comparable to SQL for relational databases.
Whenever we write our code, it is specific for the API we are writing.

Now we will see a case study.

We consider a Mobile Application used for tracking customer shipments
The following configuration information about each customer are stored in a centralized database:

- Customer name and account number
- Default currency for pricing information
- Shipment attributes to appear in the summary dashboard
- Alerts and notification preferences
- User interface options, such as preferred color scheme and font

Most of the operations on the DB will be read operations.

Naming Convention for keys -> entity type:account number
Identified entities:

- Customer information, abbreviated ‘cust’
- Dashboard configuration options, abbreviated ‘dshb’
- Alerts and notification specifications, abbreviated ‘alrt’
- User interface configurations, abbreviated ‘ui’

Document Databases: introduction

Even though people sometimes adopt document databases as if it were a relational database, this paradigma is for non-relational databases. One record is stored as a document instead of a row in a table!
Documents are usually in XML or JSON formats, which ensure a high flexibility level. Somehow, document databases recall the organization of data in OOP.
Document databases are schema-less (so that we can organize the documents how we want), allow complex operations such as queries and filtering, and some document databases allow ACID transactions as well - we can set the parameters to regulate the

tradeoff between the consistency and the availability of the system. Documents are, after all, complex data structures.
We will see that MongoDB is written in C++, it is very fast and it includes several features that you can directly call on the server. You can, of course, also write code, but it's not compulsory for many functions, which are already implemented in the DBMS.

Let's talk about XML documents. XML stands for eXtensible Markup Language. A markup language specifies the structure and content of a document.
It is the father of the structured documents, and it is deeply used in Web 1.0 - it resembles HTML, since it is based on tags. XML tags give a reader some idea what some of the data means, and it is capable of representing almost any form of information.
XML and CSS separate format and data, allowing the rise of second generation websites.
XML is also the basis for many data interchange protocols and, in particular, was a foundation for web service specifications such as SOAP (Simple Object Access Protocol - you have a client and a server; the client makes a request by exploiting an XML document in which it explains what it needs, for example the parameter of the function to be called on the server - all based on message exchanges).
XML is a standard format for many document types, including word processing documents and spreadsheets (docx, xlsx and pptx formats are based on XML).
These were the most relevant use cases.
Which are the advantages?

- XML is unicode based, it can be transmitted efficiently and opened by any editor, even if it is verbose, sometimes more than the information itself;
- It can be displayed differently in different media and software platforms;
- XML documents can be modularized, and parts can be easily reused.

An Example of XML Document

```
<item>
   <title>Kind of Blue</title>
   <artist>Miles Davis</artist>
   <tracks>
      <track length="9:22">So What</track>
      <track length="9:46">Freddie Freeloader</track>
      <track length="5:37">Blue in Green</track>
      <track length="11:33">All Blues</track>
      <track length="9:26">Flamenco Sketches</track>
   </tracks>
</item>
<item>
   <title>Cookin'</title>
   <artist>Miles Davis</artist>
   <tracks>
      <track length="5:57">My Funny Valentine</track>
      <track length="9:53">Blues by Five</track>
      <track length="4:22">Airegin</track>
      <track length="13:03">Tune-Up</track>
   </tracks>
</item>
<item>
   <title>Blue Train</title>
   <artist>John Coltrane</artist>
   <tracks>
      <track length="10:39">Blue Train</track>
      <track length="9:06">Moment's Notice</track>
      <track length="7:11">Locomotion</track>
      <track length="7:55">I'm Old Fashioned</track>
      <track length="7:03">Lazy Bird</track>
   </tracks>
</item>
```

This document shows a collection of music albums, with items, titles, artists and tracks. It does look a lot like HTML. Into tags you can insert properties, just like HTML.

XML Ecosystem:

- **XPath**: useful to navigate through elements and attributes in an XML document.
- **XQuery**: is the language for querying XML data and is built on XPath expressions.
- **XML schema:** A special type of XML document that describes the elements that may be present in a specified class of XML documents.
- **XSLT** (Extensible Stylesheet Language Transformations): A language for transforming XML documents into alternative formats, including non-XML formats such as HTML.
- **DOM** (Document Object Model): a platform- and language-neutral interface for dynamically managing the content, structure and style of documents such as XML and XHTML. A document is handled as a tree.

There are some sort of databases based on XML that let us make queries - some examples are in the list up here. XML schema, for example, specifies the attributes to put in a schema - you can modify the schema if you want to add more attributes. XML schema documents show us what we can find; XML Instance documents have to adhere to the schema and can be validated through it. Not all the attributes in the schema have to be present in the instance.

Now let's talk about the DOM: it is a representation of an XML document in the form of a tree, an additional data structure that we can exploit to navigate the XML. It is especially useful when we are using an XML document to represent a hierarchy of elements and attributes.

Usually all the elements of these ecosystems are used in more complicated systems in which they are used together. For example, in an XML database there could be a storage layer, which contains the data in a persistent way, and in which we can find the binary XML files themselves, the DOM to be exploited for navigation, and indexes.
After that there is a processing layer, which may contain an engine for the Xquery, one for the XSLT to transform the data in other formats, and a document loader. On the application layer we can use the different tools and frameworks available to interact with the database. This is a general overview.
It isn't a very used database.
In general, XML databases are platforms that implement the various XML standards such as XQuery and XSLT, and provide services for the storage, indexing, security, and concurrent access of XML files.
XML databases did not represent an alternative for RDBMSs.
On the other hand, some RDBMSs introduced XML, allowing the storage of XML documents within A BLOB (binary large object) columns.

Let's talk about the main drawbacks of XML.

- XML tags are verbose and repetitious, thus the amount of storage required increases.
- XML documents are wasteful of space and are also computationally expensive to parse - but also to transmit!
- In general, XML databases are used as content-management systems: collections of text files (such as academic papers and business documents) are organized and maintained in XML format.

- On the other hand, JSON-based document databases are more suitable to support web-based operational workloads, such as storing and modifying dynamic contents.

Just to have an idea on what can happen with XML, some years ago the university developed an application with several modules. One of them was a database with images. It was ten years ago, the database used was relational, but this is not the point. The point was that the client side generated images that had to be sent to the server side. The application was implemented using web services, so the data was exchanged by using XML files. The student who implemented the communication exploited one library directly, in which he just put the image and used XML to transmit it. This wasn't working. When they tried to deploy it in a real world scenario, where the client was generating big images. The system's memory was literally completely full, and that was because the library transformed every byte of the picture in an XML tag - the information was insanely big. A correct way would be to transmit the position of the picture via XML and use another protocol to send the picture.
It was not easy to find this error because the system was working, although not in the most efficient way. So, reading the documentation of the libraries well is the lesson to learn here.

Now we will talk about the JSON document, which is the one that we will use. We will see that in MongoDB we use BSON, which is binarized JSON, almost the same. Even though JSON means JavaScript Object Notation, it is actually fully independent from any programming language.
Thanks to its integration with JavaScript, a JSON document has been often preferred to an XML document for data interchanging on the Internet.
It allows us to format data and specify the description and the specifications of the data.

JSON example

- "JSON" stands for "JavaScript Object Notation"
  - Despite the name, JSON is a (mostly) language-independent way of specifying objects as name-value pairs

```
{"skillz": {
        "web":[
                {"name": "html",
                 "years": "5"
                },
                {"name": "css",
                 "years": "3"
                }],
        "database":[
                {"name": "sql",
                 "years": "7"
                }]
}}
```

Image extracted from http://secretgeek.net/json_3mins

An object, which is actually a document, in JSON, is an unordered set of pairs of names and values - all of them are enclosed within braces { }, there is a ":" between the names and the values, all the pairs are separated by commas.

An array is an ordered collection of values, which are enclosed within brackets [ ], and separated by commas.
In the example above, “web” is an array of objects.

A value may be a string (which must be enclosed in double quotes and can contain the usual assortment of escaped characters), a number (all numbers are decimal and use the usual C++ syntax), boolean, null, an object, an array, and values can be nested.
When we use this kind of database we might reach the highest denormalization of the data.

nested objects - in each document, really, we can include different attributes, this is just an example	nested arrays - here we have a document with different attributes and there is a field called websites, where the value is an array and the elements inside the array are documents!!
	
SIMILARITIES BETWEEN JSON AND XML	DIFFERENCE BETWEEN JSON AND XML
– Both are human readable – Both have very simple syntax – Both are hierarchical – Both are language independent – Both supported in APIs of many programming languages	– Syntax is different – JSON is less verbose – JSON includes arrays – Names in JSON must not be JavaScript reserved words

JSON is much more compact than XML!!

Let’s now talk about document databases.
First of all, they use json files and formats to store data persistently. The document is the basic unit of storage - it can have one or more key-value pairs, it might also contain nested documents, arrays, arrays of documents and generally quite complex structures.
A **collection** or data bucket is a set of documents **sharing some common purpose**. A collection is a kind of generalization of a table, the documents do not have to be of the same

type!!! I can decide to make a collection of items with different structures but that represent something similar. This means that a JSON database is schema less (predefined document elements must not be defined) and it has a polymorphic scheme (in a collection, the documents might be different).

We can start from an empty database, just setting up the parameters of the server. While running the code, we can create the collections and populate it.
If the database is relational, the first thing we must do is define the scheme, implement it, create the table, and after that we can modify and add content to the tables. Ok, in SQL schemes can be modified, but it is not a good practice.
Document databases do not require this formal definition step. Developers can create collections and documents in collections by simply inserting them into the database.

What are the pros and cons of being schema less?
One of the pros is the high flexibility in handling the structure of the objects. We can add, modify elements and make them coexist.
One of the cons is that the DBMS might not be allowed to enforce rules based on the structure of the data. Controls should be done on the application side, because the DBMS itself can't do so much about it.

Another consideration: if we want, we can use MongoDB as a relational database. A document database, theoretically, could implement a third normal form schema: tables can be simulated considering collections with JSON documents with predefined structure. It is a bad idea, because we must arrange all the checks manually. We would be using a tool in the wrong way, spending time reinventing the wheel and risk making several errors.

Document databases usually adopt a reduced number of collections for modeling data.
Nested documents are used for representing relationships among the different entities.
Document databases do not generally provide join operations.
Programmers like to have the JSON structure map closely to the object structure of their code!!!

Document Databases: data modeling and partitioning

Document Embedding is one of the solutions proposed in order to replicate a kind of relational schema inside of a document DB. The example in the slides embeds an array of actors inside the document describing the film - this represents a relationship between these two entities. This solution allows us to retrieve a film with all the actors - the actors will be duplicated across multiple documents though.
So, before structuring the documents we must look at the use cases of the application and only use document embedding if it is very convenient.
In complex designs this solution could lead to issues and inconsistencies - moreover, some JSON databases have limitations of the maximum dimension of a single document.

Another approach is called Document Linking. It is most similar to the relational approach. We still maintain the flexibility of not having schemes. We embed the IDs of the documents

inside the other document, and the IDs can be used to retrieve the other documents if needed. We might also add other few pieces of data in order to give a preview, if we add redundancy. We need two queries, one for the first and one for the second document. Now, at least two collections of documents must be defined.
The biggest error is to roll back towards the third normal form, if we use a collection for each table. This is why the use case diagram is important: we do not want to use another kind of architecture just to treat it like a relational database!
Document linking approaches are usually somewhat an unnatural style for a document database.
However, for some workloads it may provide the best balance between performance and maintainability.
When modeling data for document databases, there is no equivalent of the third normal form that defines a “correct” model.
In this context, the nature of the queries to be executed drives the approach to model data.

Writing down the requirements, interaction of user and application, and use case scenarios are fundamental to build and model the data.
Requirements expressed in natural languages must be translated in queries.

Some general rules for modeling relationships.
If we want to model one to many relationships, and if we need to exploit in our application the fact that many objects are linked with one, we may exploit the document embed.
The basic pattern sees the “one” entity in a one to many relationship as the primary document, and the “many” entities represented as an array of embedded documents.
Document embedding, as we already said, is only useful when the two entities are always looked up together. Otherwise, document linking is a better approach - or even an hybrid approach, where there is a preview.

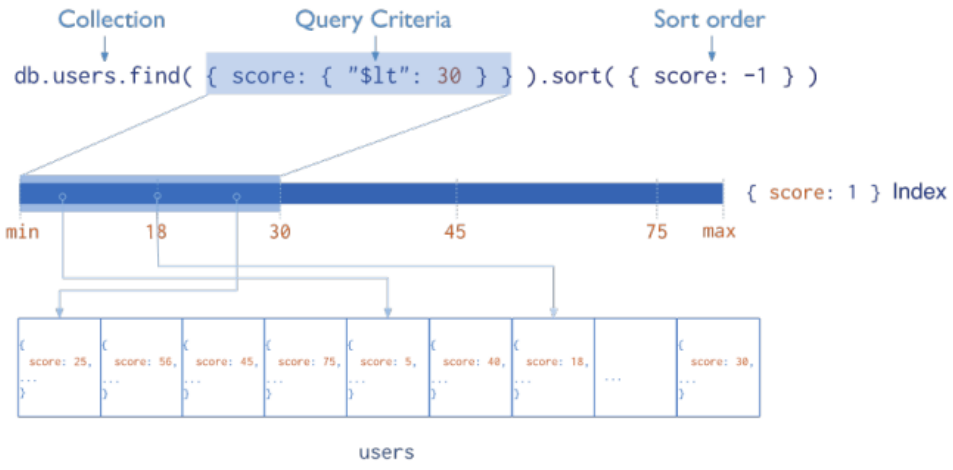
How do we represent many to many relationships?
For example, a student can be enrolled in many courses and a course can have many students enrolled in it.
We can model the situation in a document linking approach: having a list of students’ IDs in the courses documents and a list of courses’ IDs in the document of the students.
The first problem is that, on the side of the application, we must take care of the referential integrity - the DBMS will not control it.
We still have the properties to consider that we do not have a fixed scheme.
An example of a many to many relationship depends on the query we want to do, of course.

Sometimes we have objects organized into hierarchies. There are different ways in which we can organize them.
What if we want to show frequently the specific instance of an object, and then show the more general type of the category? In this case, we use the parent reference solution.
That is to say, we have a collection in which we put the specific object, which has an ID, the name, and the ID of its parent. We can go on like this until we get to the root, and we will look at the tree from a bottom up approach.
What if we want to retrieve the children of a specific instance?
We specify, from the root, the IDs of the children, and so on.

There is also another solution, exploiting directly the list of ancestors - this solution allows to retrieve the whole path of the ancestors with one read operation, just putting the IDs of the ancestors in the document of the child.
One problem with this is that a change of the hierarchy structure might require many write operations - how many also depends on the level at which the change occurred.
The solution on what is best to do depends on what you want to do with the data.

Let's see another scenario. A company has a fleet of trucks and they must send to the database some information about the truck itself every three minutes.
So, we have a relationship 1 to many between the truck and its information.
In a working day, for each truck we have 200 records of information (if this info contains truck id, time, date, driver name and fuel consumption rate, some info during the day will be repeated many times).
A more effective solution would be to organize a document for each day, with info about the trucks and for each truck a vector with all the non repeating information.
In this case, embedded documents make sense also because we use them for statistics.
Attention: we can have a potential performance problem, because one document for each truck for each day… clearly we allocate the space for the document when we create it. The DBMS allocates a certain amount of space, but if the document grows more than expected, the DBMS has to relocate the data to another location, and free the previously allocated space.
Which solution can we adopt here? If we already know the number of documents to insert in the array, we can allocate them at the beginning!
In conclusion, we have to consider the life cycle of a document and plan, if possible, the strategies for handling its growth.

Now, some words about indexing data in a database - it is important in order to avoid scanning the whole database.
Indexes, like in book indexes, are a 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, put in a precise order determined by the value of the field.



This index is a data structure with pointers towards addresses - it is ordered and the order must be maintained.
Indexes are very nice for read-heavy applications - things with analytics and many queries - because they allow the user to quickly access the needed data.
What if we have a write-heavy application? The higher the number of indexes, the higher the amount of time required for closing a write operation: this is because all the indexes must be updated.
This is another trade-off: more indexes means faster reads and lower writes, and vice versa.

We can also build a more complicated system. In this case, in a unique application, we have the data analysts that makes analysis of the data in the Document DB tuned for reads - called the OLAP side, the OnLine Analytical Processing.
These data is produced on the Document DB tuned for writes, called also OLTP (OnLine Transaction processing) which is a write heavy database. We can suppose that the analytics are not extracted in real time, so between the OLTP and the OLAP there is an extraction, transformation and load process. Of course, the information inside the two might be different in format and organization, while regarding the same process of course.
Here, we do not need indexes.

Vertical partitioning is a technique used in relational databases for improving the performance when some data does not need to be retrieved. It is done by separating columns of a relational table into multiple separate tables that are split in different parts of the disk.

This introduces the concept of Sharding, also called horizontal partitioning, which is the process of dividing data into blocks, or chunks. Each of them is labeled as a shard and 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.

What are the advantages of sharding?
It allows us to handle heavy loads of traffic and users. Data may also be easily distributed on a variable number of servers, and every server may be added or removed by request.
It allows for horizontal scaling, which is much cheaper than vertical scaling.
Combined with replications, it ensures fast responses plus a high availability of the system.

How do we implement sharding?
We must select two things.
The first one is a shard key - that is to say, one or more fields that exist in all documents inside a collection. This field(s) will be used to separate documents, and in order to do that we can select any atomic field in a document.
The second thing are partition algorithms.
There are three main categories:

- 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's 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.

Document Databases: design tips

The main feature of this type of database is flexibility.
Our main concern today is to think about how to insert data inside collections.
Collections are sets of documents - like tables in relational databases or buckets in key value databases. Thanks to the polymorphic approach implemented in this kind of database, we can store documents of different types in a collection - in general, collections should store documents regarding the same type of entity.
Not exactly the same type but documents which share some common context information.
How can we discriminate what entity is described by what document?

```
{ "id" : 12334578,
    "datetime" :  "201409182210",
    "session_num" : 987943,
    "client_IP_addr" : "192.168.10.10",
    "user_agent" : "Mozilla / 5.0",
    "referring_page" : "http://www.example.com/page1"
}
```

web clickstream data

```
{ "id" : 31244578,
    "datetime" :  "201409172140",
    "event_type" : "add_user",
    "server_IP_addr" : "192.168.11.11",
    "descr" :  "User jones added with sudo privileges"
}
```

server log data

These two documents have some similar and some different data. Are they two entities or two instances of the same entity? Can we store the two documents in the same collection?
It depends on how we are going to use this data.
If we store the documents together - we consider them instances of the same entity.
First of all, we must add a new field to the documents, which is the doctype, to discriminate if we talk about a click stream or a server log.
When we store data on the disk, it will be stored in the same section. This situation can lead to inefficiency if we use the information separately, whilst if we use the information together it's good.

We can use indexes - maybe on document type.
recall that they reference a data block that contains both clickstream and server log data, the disk will read both types of records even though one will be filtered out in your application.
If we have a large collection with a huge number of different types it will be faster to scan the entire document than the index.
Store different types of documents in the same collection only if they will be frequently used together in the application.
Maybe they are used together in queries and analytics on common fields.

We can also recognize whether we are doing well or not by looking at our code. When we work with NoSQL, our code will have a higher number of lines and functions.

In general, the application code written for manipulating a collection should have:

1. A substantial amounts of code that apply to all documents
2. Some amount of code that accommodates specialized fields in some documents.

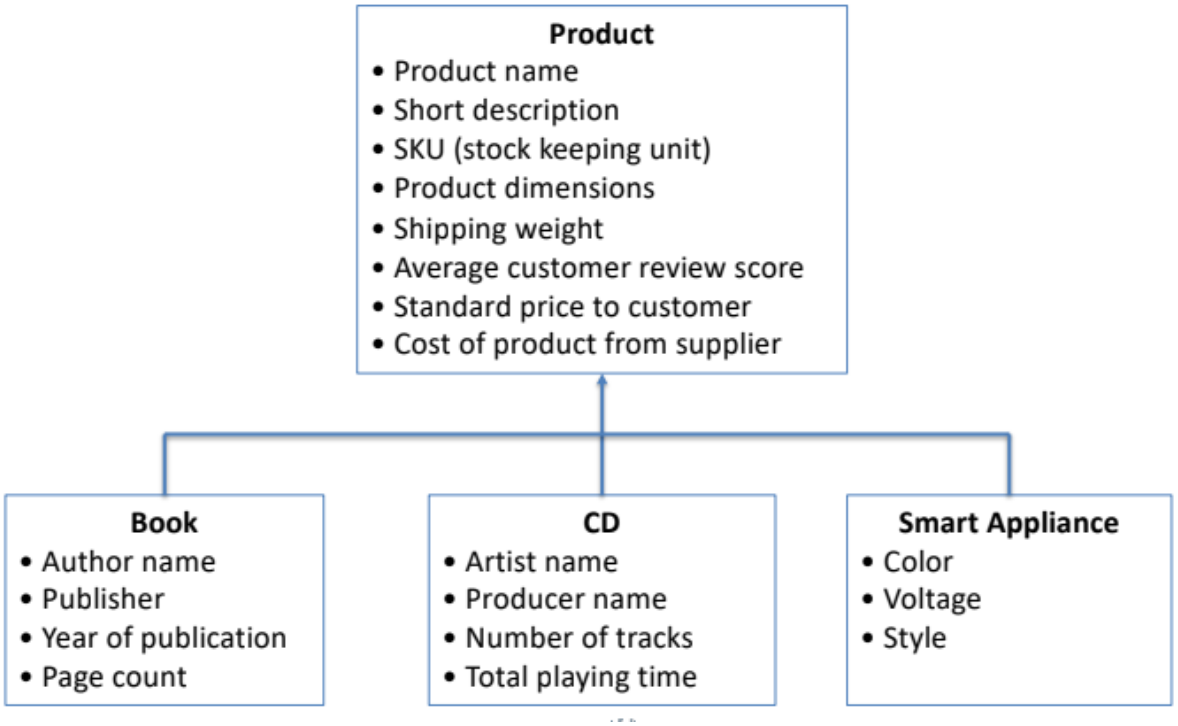
```
High-Level Branching
   doc.
If (doc_type = 'click_stream'):
    process_click_stream (doc)
Else
    process_server_log (doc)
```

The case of High-Level Branching like in the picture, where the code separates at the level of checking the type of document, can indicate a need to create separate collections.
Branching at lower levels is common when some documents have optional attributes.

If the branching is only done for some attributes it's ok.

This is the case when we have a class and then a derived class with some more attributes.

So, let's suppose that we have this type of data.



So, we are describing a product which is an abstract class that can be specified in Book, CD, SA. Let's suppose that we have the e-commerce application and the document database, and we must organize the data.
In the relational database we only have to do the normal form and that's it.
With the document database we could have a different number and disposition of collections. There is no right answer as of now, we must first see the requirements.

So, first of all we look at the queries that our application must be able to answer:

- What is the average number of products bought by each customer?
- What is the range of number of products purchased by customers

- What are the top 20 most popular products by customer state?
- What is the average value of sales by customer state
- How many of each type of product were sold in the last 30 days?

So, should we group them all together or not?
Do we have any specific query that only asks about a kind of product? No, products are always looked at generally, so we can group them together. It's better. otherwise, we would have to query three collections every time, at least for the products.
Notice that: If we separate the product into different collections, and the number of product types grows the number of collections would become unwieldy.

At the end of this discussion we ask ourselves: do we have to use

- normalization
    - more collections
    - avoids data anomalies
    - causes performance problems
    - require join operations
- or denormalization
    - less collections
    - more redundancies
    - improved performances
    - avoid join operation
    - supports improving read operations when indexes are adopted

???
In the query that we are writing, there should be as few collections as possible, to avoid joins and to keep everything denormalized.

Column Databases: introduction

As far as this paradigm is concerned, we will only see the theoretical part, the modus operandi on how to work with this. We do not have time to see Cassandra in action. We are still free to adopt this in our database.

But first, an introduction of the first column databases, not distributed ones.

The starting point towards databases organized in column sees that we may have two main activities on our complex information system - on one side OLTP, on the other OLAP.

OLTP - Online Transaction Processing: software architectures oriented towards handling ACID transactions. Imagine banking or supermarket transactions.
OLAP - Online Analytics Processing: software architectures oriented towards interactive and fast analysis of data. Typical of Business Intelligence Software. This is more for managers and analysts.

In general, we must consider at least two servers, or two group of servers, one for OLTP, one for OLAP.
OLTP has these features:

- High volume of transaction
- Normalized data
- Many tables
- Fast processing
- Queries look like “who did this”

OLAP has these features:

- High volume of data is analyzed
- slow queries
- denormalized data
- fewer tables
- Queries look like “how many people did this”

THis is the situation with which we start. In relational databases the high level organization in terms of data and tables was projected into an organization by row on the disk - so one record was in a row on the disk. One row of the table was in a sector.
So, when we talk about OLTP processing there are no problems with this organization, one transaction inserts a record in an atomic way. We work mainly on rows, it works well at a disk level. In an atomic workflow, the first relational databases, record oriented in terms of workflow, together with the physical structure structured in rows, OLTP performed well.

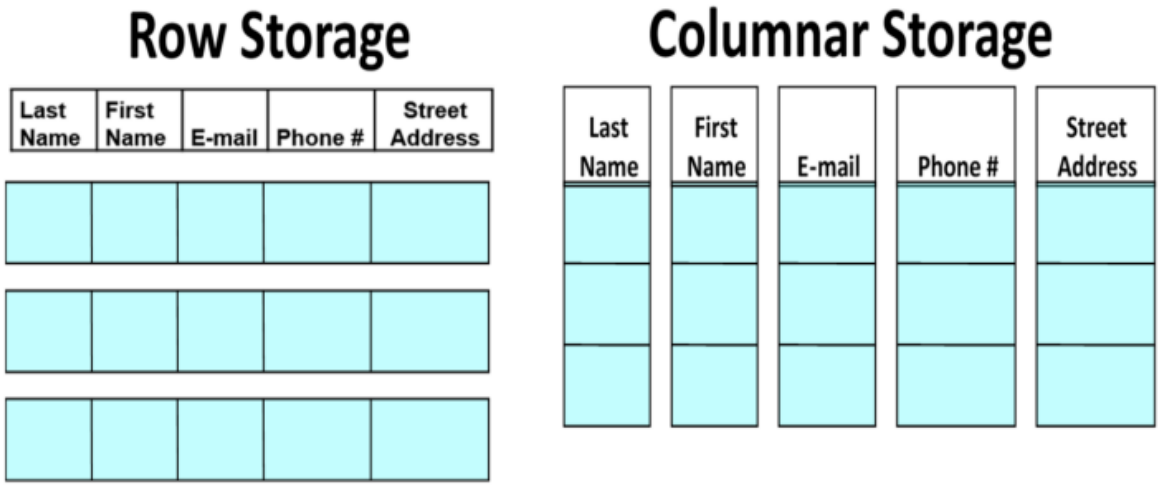
We are talking about a record based processing era, before the one we are experiencing now. In this era the most critical operations were CRUD operations.
On the OLAP part of an information system we were allowed to run reporting programs in a background batch mode, with no problems scanning the tables row by row and reporting analytics. The time needed for calculating average values of a company’s activity was not a problem.
We started having performance problems when the business intelligence software became even more important for the user and the companies - the importance of having reports and statistics more frequently.
In this period, there was a problem with working with a DBMS structured with a strong normalized collection of tables like a relational one. So, from one side writing and reading from the database was very easy, on the other hand analytics were difficult.

For this reason, in this period data warehouses appeared. These are information systems in which relational databases were involved to support the analytics and the decisional part of the processing. So, the OLAP part was supported by another DBMS - organized in an appropriate way, based on a relational database, mainly devoted to offering analytic and decision services.
This was very important to have a separation between the two worlds.
Please, note that in OLTP we talk about operations, out in the field, we produce transactions, actions; in the OLAP side we produce information that can be used in the OLTP side to change the actions maybe!
OLAP, as we said before, can operate overnight even. We do not worry too much about there being a strict consistency between what is happening in OLTP and in OLAP. The data that goes from OLTP to OLAP gets organized in an according way, even overnight, and is processed. What is produced goes on to become a business strategy.

Let's see what was one of the strategies adopted in a data warehouse in which we had relational databases and their theory available.
Star schemas in data warehouses are databases composed of tables in which we have a central table, called the table of the facts, which has a huge amount of data, up to millions of records. In this table we distinguish between the attribute of the facts and the dimension of the attributes.
The fact about an event shows us what we can represent from this event.
Then, there are many smaller dimension tables. These are all pieces of information that add to the fact.
Why do we use this scheme? Because the aggregation query can be executed very quickly thanks to this separation. Doing statistics of the dimensions becomes very easy.
Star schemes do not represent a fully normalized relational model of data, there are some redundancies and information sometimes depends partly on the primary key.
Another example in which we can exploit a star scheme is meteo data.

Which are the drawbacks of the star scheme?
Data processing in the data warehouses stayed intensive both on CPU and IO.
The amount of information kept growing a lot and users demanded more interactive response times.
The discontent, during the mid 90s, increased.

Other solutions had to be thought out. Indeed, if we ask for analytics, for example, we would like to know the average value of incomes each day in a supermarket, what do we need? We need a column. Or for example, we want to calculate the age distribution of the students in this university - we just need the age in order to do this.
Generally speaking, when processing data for making analytics, in most cases, we are interested in retrieving the values of one attribute of a set of records and not of all the information.
If we are dealing with row based storage of records, in order to retrieve a row we have to access the whole row.
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.



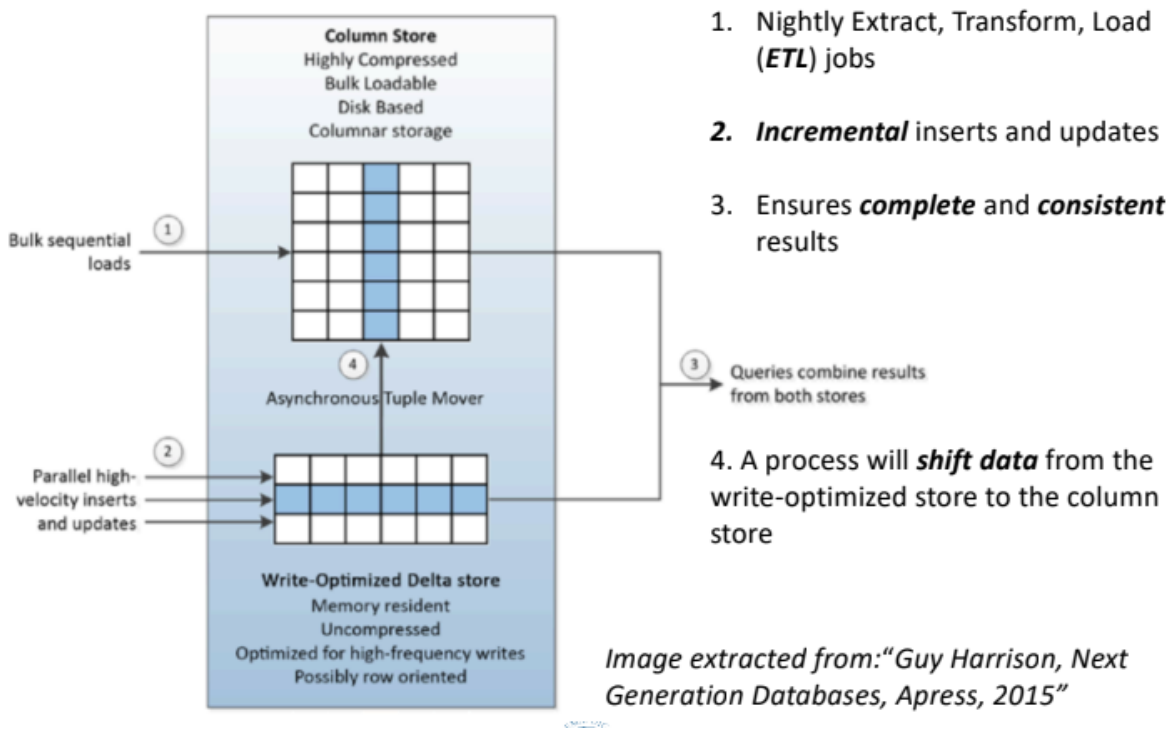
This is made possible by the columnar storage.

Now, let's suppose that at a high level the tables are still organized like in a relational database, but on the disk data is organized in a columnar way.
If data is stored by rows, in order to retrieve the sum of salaries we must scan many blocks.
In the case of column storage, a single block access is enough.
In general, queries that work across multiple rows are significantly accelerated in a columnar database.
Obviously, the amount of cpu and io activity will depend on the single activity, this only works in general.

If we exploit the columnar organization of the data at disk level - but also at a high level - we can also exploit another aspect of data theory: compression of the data. Data compression algorithms basically remove redundancy within data values in order to reduce the dimension of the data.
The higher the redundancy, the higher the compression ratios.
To find redundancy, data compressions algorithms usually try to work on localized subsets of the data.
If data are stored in column databases, a very high compression ratio can be achieved with very low computational overheads.
Usually, data stored in a column is also sorted - this means that we do not really need to represent the whole data, but just the first item and each item as the delta from the previous one.
Most of the systems based on this compression are also able to introduce low overhead when data has to be retrieved.
However, column databases perform poorly during the execution of queries that only act on a single row.
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 more than once blocks of the disk.

Now, the problem of respecting consistency should be taken into consideration. Some clients want to have analytics based also on info coming at high speed from the OLTP side - data warehouses have some limitations here, since writes are handled mainly on rows whilst analytics prefer columns. What can we do now? We may remove the columnar part and go back to the previous organization?
Some productors proposed the Delta store as a solution.
It is one area of the database in memory, not compressed and optimized for high frequency data modification.
Usually, this area is row-oriented. So on the disk we have a columnar organization. In the memory, we use a row organization.
So, from one side we have the user that asks for analytics; on one side we have the production of aggregated data at night or each tot time - and we also have the possibility to use real time produced data.
So, the data from the delta store are periodically merged with the main columnar oriented store.
Queries might need to access both information in the column store and in the delta store.

In the following picture we have a scheme of the architecture.



The data in the columnar storage in the disk is very compressed and sorted. Bulk loadability means that using an ETL process we can load the data in huge groups.
We might need to exploit other transactions in real time - like insert and update. This can be oriented directly towards the delta store in an incremental way. So the steps are four, we can see them up here in the picture. First we build, during the night, the columnar database. Second, during the working time we might accept other data - third, queries unite the two data results. Fourth, periodically we put the data from the delta store in the disk.
This is still a talk about the past.

We may have another type of problem when working with data warehouses and exploiting, for example, delta stores.
The problem regards the fact that when we make complex queries we need to consider information coming not only from one column, but even coming from different dimensions of the data we are analyzing.
So, we think about putting together on the disk also the columns that will be retrieved together. But, nobody will insure us that two different dimensions will be stored together. One solution was to exploit projections, which are combinations of columns that are frequently accessed together and thus are stored together, or closer, on the disk.

Imagine having a classical table, organized in rows: “Region”, “Customer”, “Product”, “Sales”. In the disk we organize everything in columns. What if we want to put together Region and Product, or Customer and Sales?
We need to access more sections of the disk, and we are not sure whether they are close or not.
The default storage is called “Super Projection" - a simple organization with unsorted columnar storage, and it is exploited when we make analytics on a single column.
If we want to make a query like I said before, the system can organize in the same section of the disk the dimensions that we need, and sort the values as well.

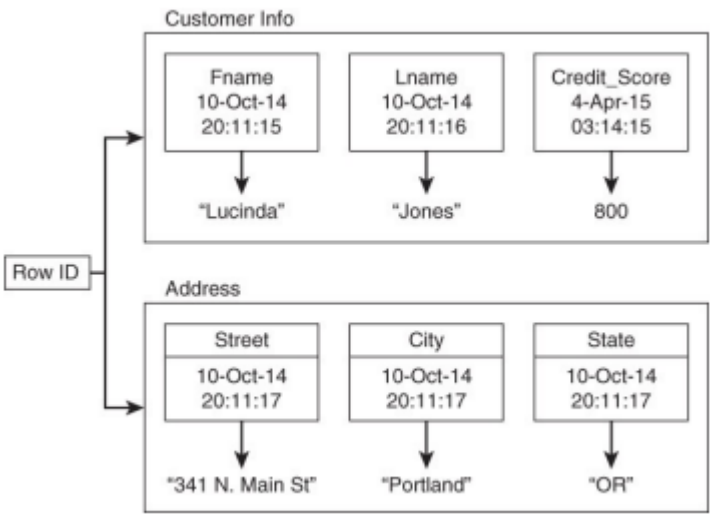
We have redundancy but now we can support queries more easily!
There was a DBMS called Vertical that exploited this organization.
One question is: how are projections created? There are some tools to create them manually by the admins, or advanced tools which create projections directly based on the queries proposed by the system.

Oracle proposed a hybrid columnar compression solution, since we might need to modify rows in any way, and also support analytics on the attributes. The idea is that at high level data is organized in units of 1Mb. In each unit we have the rows, so fast access towards the rows contained in the block. Then, inside the block we have a columnar organization of 8k each columnar block.
This solution allows us to make modifications on the rows and analytics on the columns.
If i want to modify a row i have to identify the block and then finding the row is easy.

Columnar Databases: details

There are some issues that we may have when we talk about key-value and document databases.
The main problem with key value databases is that to adopt them we must have a very simple scenario, they are not designed to work with data with many attributes that should be retrieved and used together.
Document databases have a different query language, different than SQL. Most other solutions can be handled via SQL.
Columnar databases may offer some features found in these other two, but they are a bit closer to the functioning of a relational database in terms of usage.

BigTable was one of the first NoSQL databases and the first columnar one. It was introduced by Google in 2006 to support Google Earth and Google Finance.
They had to handle very big data, so they needed flexibility and availability. This db is based on tables composed by columns, a columnar organization. If we want to identify a record inside the database, and maybe even just one value, we need the ID of the row, the name of the column and the timestamp. These are the three indexes of the values.
When we make an operation, the level of atomicity regards a row, and rows are maintained in a certain order.



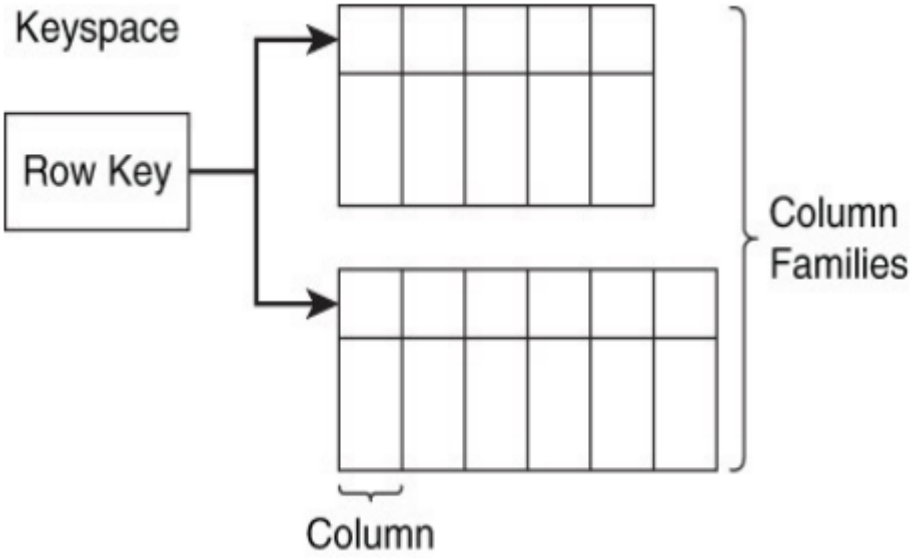
Here in the picture there is an example. One element in a row is described in terms of column families. A column family is a collection of columns with some column properties, such as that they are frequently used together. In the picture two column families are Customer Info and Address.
The columns that are within a column family are usually kept

together on the disk. Applications may add and remove columns inside families, like key-value pairs, but column families must be defined a priori. This means that this system is not fully schemeless, since the families must be decided like tables in a relational database.

So, one record is identified by a row id. We must identify the family attributes we use at the beginning but the attributes to put inside can be changed and decided later on.

So, to extract a single data from this database we must specify:

- Row identifier (like a primary key in a relational database)
- Column name (which identifies a column, like a key in a key-value database)
- Time stamp (in order to retrieve one specific version of the data)



In general, whenever we work with this kind of database we must define a Keyspace, that is to say, a sort of equivalent of key-value's namespace.
It is the top level logical container and usually we have one for each application.
It holds column families, row keys, and related data structures.
It is a collection of the elements that we consider in the application from the side of the database.

Row keys are the components that allow us to uniquely identify a row stored in a database.
At low level, data is organized in a column approach though.
So, with a row key we identify the record - the record itself is split in several column families.
The row key can be used to create shards and to order data - algorithms for data partitioning and ordering, though, depends on specific databases.
As an example, Cassandra adopts the partitioner, a specific object, which is used for sorting data. Moreover, by default, the partitioner randomly distributes rows across nodes.
We have to keep in mind that this is only a logical point of view.

Columns have the same role as a key in a key-value database. So, the row key can let us identify a specific entry if together with the column key.
The time stamp or other version stamp is a way to order values of a column.

There are some frameworks which let us define data types (Cassandra), some that don't (Big Table).

Columns that are frequently used together are grouped into the same column family.
Column family analogous to a table in a relational database. However:

- 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.

In the same column families we can mix data with different types of values and formats.
We can also specify other attributes or insert new entries dynamically with no problem.

The point of strength of a column family regards the fact that we can order and organize values; we have versioning thanks to the timestamps, and we can modify and add columns from the application side no problem.

Columnar databases exploit flexibility even though it can be a bit reduced by the fact that we must set the column family. We supply this problem by being able to produce statistics and analytics very quickly.

What about the distributed organization?
We can distribute the data on clusters to ensure availability, mainly using replicas, and scalability. Consistency can be a problem but we can exploit eventual consistency.
Thanks to the partitions, we can have the system balance the load.
So, from a very simple point of view a partition is a logical subset of a database.

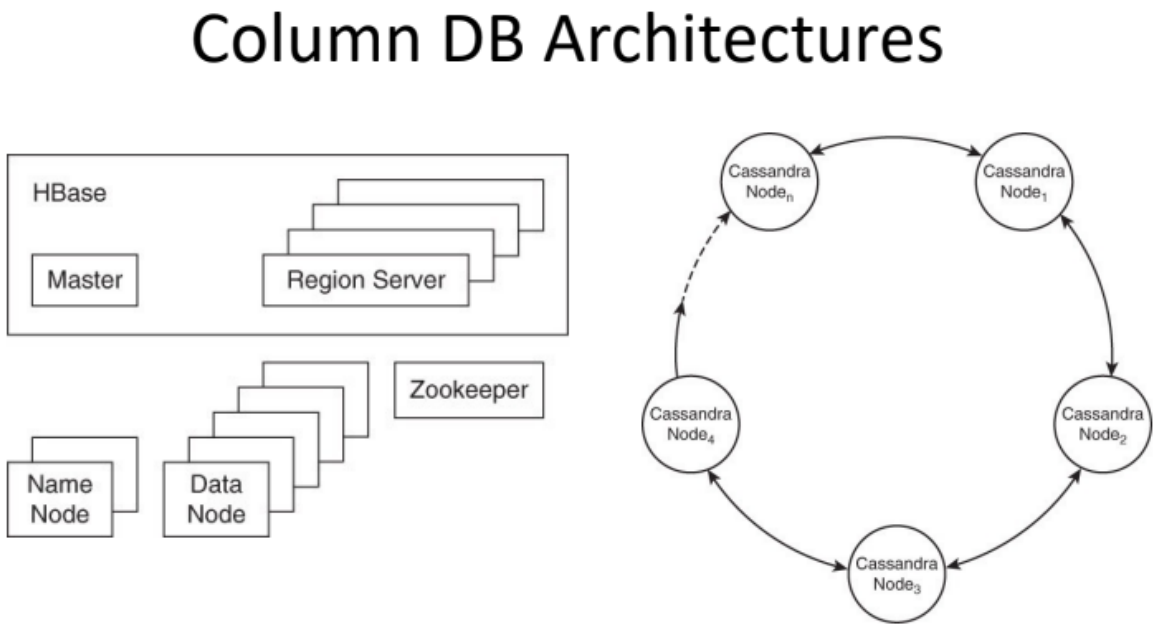
Partitions are usually used to store a set of data based on some attribute of the data in column databases, they can be generated on the basis of:

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

Some columns can be distributed on some servers and some on others. The most important thing is that we do not have to access different servers to take information from the same column family.
We can partition data both horizontally and vertically - it depends on the framework that we use.

The architectural part: two kinds of architecture may exist, in general.
The one on the left is the architecture of HBase, Apache's version of Big Table.
The Apache Project is an open source license with many frameworks.
HBase exploits the Hadoop environment, so at a low level it is based on HDFS.
It has a master slave organization, the slaves are called region servers and they are the nodes in charge of managing the user requests. A region server may include a collection of regions, and one region may contain all the rows of some specified keys - so one region is a partition.
So, at the level of the region server we have data structures devoted to the organization of the data and of the namespace. They are deployed on different servers.

Then we have the zookeeper, which is a node that coordinates the communications between all the other nodes.



Apache HBase Architecture ***Cassandra peer-to-peer architecture***

The key point is that this HBase architecture is characterized by the fact that we have different servers, and each one is specialized by a specific function, a specific collection of services to support the entire ecosystem.
We have more points of failure.

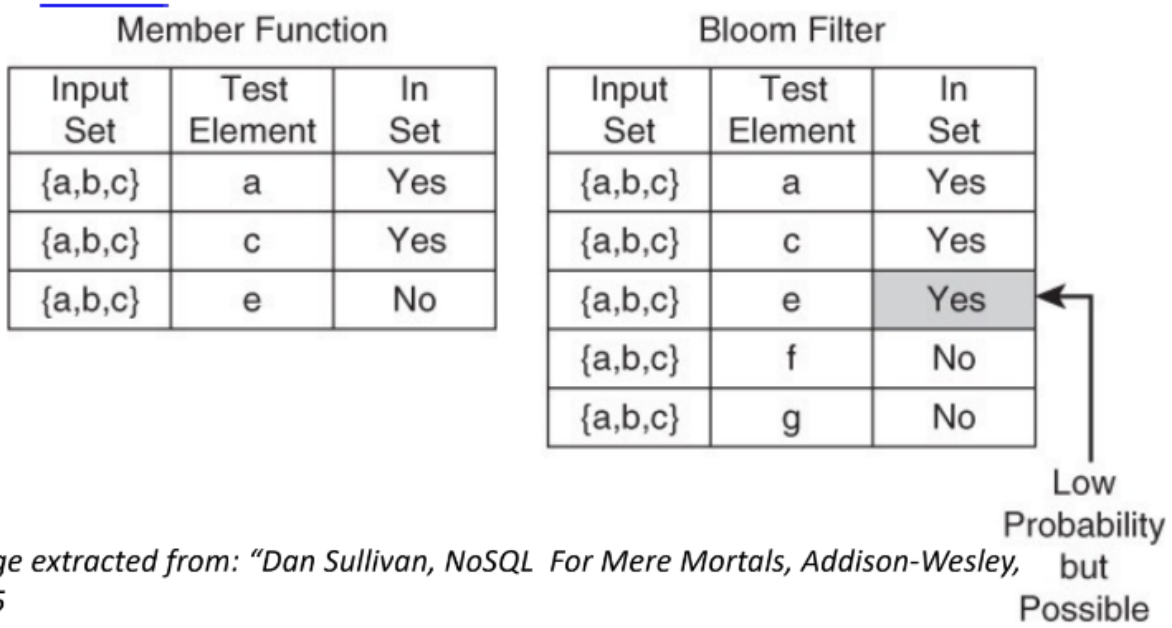
On the other hand, we have the Cassandra peer-to-peer architecture. Each node here can run the same code, the same program. Even if on one node we have specialized software for some services, we can run the software on each node. We do not have SPOFs.
This organization is very useful for horizontal scaling.
We can have replicas, data partitioning on different nodes.
Now, let's see a collection of data structures and strategies that we can use in distributed databases, and that are used for sure in Cassandra.

First strategy to speed up the write operation on a server. The idea is that if we make a write operation on a server we must wait the time of storing data on the disk, and on the disk we are not sure that the data is stored in an adjacent position.
so, once we wait for this time we obtain the acknowledgement from the side of the application and we know that the data is permanently stored.
To speed up this process there is a data structure that we can use, called Commit Logs. These are append only files (we can only write at the end) that, if used, make sure that all the information is stored in the same sector of the disk. So writing in the commit log is faster than writing on the specific column family.
So, what happens:
We make a write operation. We make an entry on the commit log that puts all the details about the write operation. Then we return the control to the application. Then, a thread will copy the info from the commit log to the persistent storage.

They are also used to ensure a high level of control of the failures.
If while we are updating the database we have a failure, the logs show the point at which we were, and we can also recover the information that we may lose.
So, there are commit logs in Cassandra. They are simple append only files that exploit the situation in which data is stored in the same place on the disk. It is one unique file.
This is one of the typical questions on the exam.

Bloom Filters are a general concept of a probabilistic data structure, used as support for a read operation. When we make a read operation we need to make an access to the disk, and we may have a certain latency. We may have a get function on one row key, a specific column, and timestamp, and we are not sure that we have the data on the location: this means that we might make an access to the disk without retrieving the data.
In order to minimize useless accesses towards the disk we can use this data structure, and its functions.

It works like this: we can make a query towards the data structure, that can be also in memory, and the query is “do you have this data?” The answer is “no” or “maybe”. If it says no, then the data is not in the part of the memory we are considering. “Maybe” might be a false positive. False negatives are not allowed.



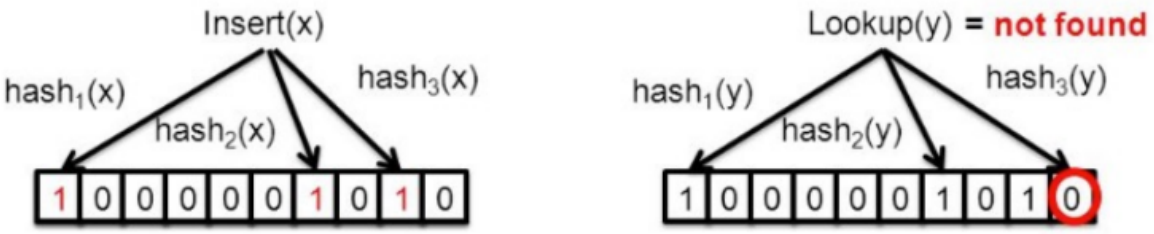
age extracted from: “Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 15

If we use a member function, like a hash table, we can say for sure whether something is in the input set or not.
In the Bloom filter, we can say “no” for sure. We cannot really say “Yes”, we can say “there is a probability that the data is not present or present”.
We can have true negatives, true positives, false positives. No false negatives.

How can we implement a Bloom Filter?
Let’s suppose we want to build a Bloom Filter for a partition - we want to know if that specific shard contains or not a piece of data. So, we make an access to one server, and the server may identify the partition on which the data should be stored. Before accessing the server we want to check whether it could be there.

We could build an array with N binary values. At the beginning all values are set to 0. Then we identify a collection of k << N hash functions. These are used to map each key, each data that we are looking for, into a position of the vector.
Imagine that we have the row id, the column, the timestamp. We pass everything into the hash function, do the module and watch at the position in the array.
Everytime we have new data, we set the corresponding value in the array to 1.
When we make a read operation, we calculate the k hash functions, look at the indexes, and we will state that the data is present if and only if all the corresponding indexes are equal to 1. We are not sure that we will find the data in memory, even in this case.

- 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



Suppose to have a Bloom Filter with N = 10, and 3 hash function.
Suppose that we want to write X. We calculate the hash values and we write 1 on the indexes.
Suppose that we want to read Y. We calculate the hash values and check the values of the indexes.

Important property of Bloom filters is that the time it takes both to add an element and to check if an element is in the set is constant.
The probability of a false positive decreases as n increases, while it increases as the number of elements stored in the database, or if many elements are canceled from the database.
Removing an element from the simple Bloom filter is impossible because false negatives are not allowed.
If you want to flush, you flush everything!
One example of question for the exam: “which are the strategies for when we want to reduce latency during read and write operations?” the correct answer is “Bloom filters for reads, Commit Log for writes”.

Suggestion: use n = O(h) and k=ln2(n/h)where h is the number of value to check against the filter.
K can be calculated as the log of the fraction of N over h.

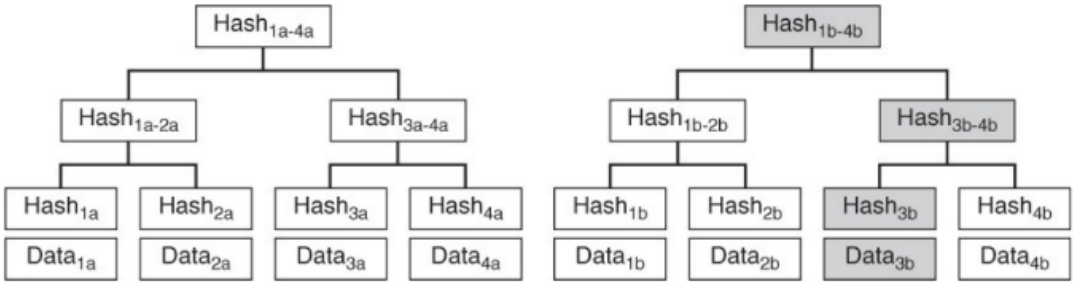
Now let's make some considerations regarding replications.
Replicas allow us to ensure the availability of the system.
Also, for this kind of databases we can have different levels of consistency. Consistency level refers to the consistency between copies of data on different replicas.

- 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.

When we have to ensure a certain consistency level, we must remember the replication feature of the database, which should be somehow controlled. We will see one strategy for checking whether the copy on one server is updated with the one in another server.
Usually, in this kind of column based databases, we have the logical ring organization. One server acts as the most important replica and it is identified by a hash function, and the first server is in charge of implementing the consistency strategy.
In Cassandra the replicas are stored in successive nodes in the rings in clockwise direction.

Anti-entropy is the process of detecting differences in replicas.
So, the simplest way to implement it is to check element by element if there are differences, but it is very heavy computationally. It is important to detect and resolve inconsistencies among replicas exchanging a low amount of data.
One simple way to implement this is by using a hash tree. It is a binary tree whose leaves contain the hash value of the collection of a block of data.

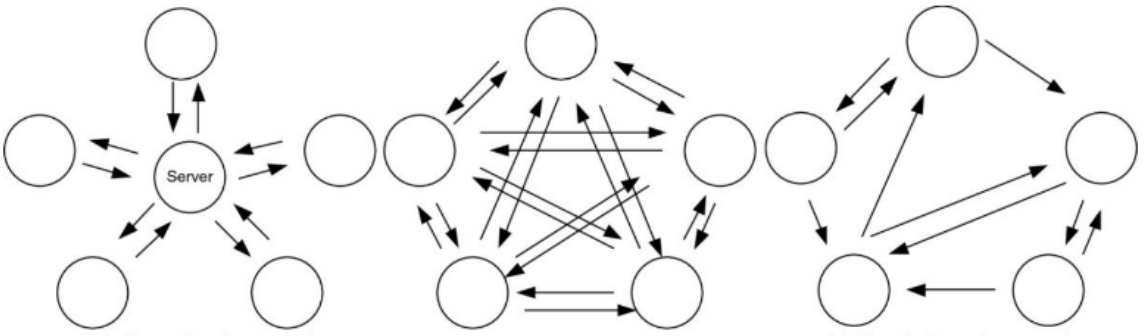


Then, at each level, we have the hash value of the hashes of its child nodes. So, we can calculate the hash value by putting together the hashes of the two nodes under it.
The root contains the hash value of the whole partition stored on one set.
So, it is a sort of compression of the information of our database.

If we have differences among the old tree and the new tree made on the same data, we can track where the differences are, by starting from the root and tracking the different hashes going down, until we find the partition in which differences are identified.
So, transfering the tree and examining it is a fast process.

One important aspect in order to implement partitions and replicas, and update everything, nodes must be aware of each other's conditions. So, they must communicate in some way. This can be achieved in many ways. The simplest one is to let nodes talk with each other. This works with a few nodes - but while the number of nodes increases the number of communications grows in a quadratic fashion.

There are many different approaches to save some time.
Here we have three typical approaches.



(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

The central approach is the fully connected overlay approach. We've already seen that it is not scalable.
The centralized approach creates a single point of failure.
The third approach, called gossip-based approach. It works a bit like how gossip spreads in real life. One node talks about himself and other nodes to other randomly selected nodes.
They operate in parallel.

How does the architecture of Cassandra work for read and write operations?
Every node can handle client requests. One node answers to your request - any node. Any node knows where the information you are looking for is contained. If one node is out of service, other nodes can take its place.
In the READ case, you make a request, Node 1 answers but the information you are looking for is in node 3. Then Node 1 will forward the request to Node 3, which will answer to Node 1, and Node 1 will answer back to you.
What about write operations? It's the same. We make a write request, one node answers and, if needed, forwards the write operation to the right node. If a node is not available for any reason, the first node maintains the request in a commit log.
This is called Hinted Handoff: it is a process in charge of storing info of the write operations, checking periodically the availability of the server that should be updated, then updating the information when the server is available. We avoid losing data, this way.
It:

1. Creates a data structure to store information about the write operation and where it should ultimately be sent
2. Periodically checks the status of the target server
3. Sends 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.

So, Commit logs are useful to speed up write operations.
Bloom Filters are useful to speed up read operations by checking probabilistic data structures in order to minimize useless disk accesses.
Anti Entropy, for updating replicas by using a hash tree and exchanging hash trees and comparing them.

Communication protocols: gossip protocol.
Cassandra read protocol, and write protocol. So many important things!!

Column Databases: design tips

Some guidelines for designing this kind of architecture. In all the kinds of architecture that we have seen and will see, the starting point is the collection of requirements. Users and requirements, and queries, guide the DB design.

In the following, we show some examples of typical queries that may led to choose column DB:

- How many new orders were placed in the Northwest region yesterday? (histogram-like data)
- When did a particular customer last placed an order? (versioning)
- What orders are en route to customers in London, England?
- What products in the Ohio warehouse have fewer than the stock keeping minimum number of items?

All of these queries are sort of analytical queries. They mostly can be represented by graphs, even.
When we design this kind of database, we must give special care to the attributes. So, when we design such a database, we need to abstract information regarding:

- Entities: we must recall that one instance is a row uniquely identified by a row key - we have a strong denormalization but still have tables;
- Attributes of entities: these are modeled using columns - remember that we must determine which will be used together or not;
- Query criteria: determining which queries will be most requested and used will determine the optimal way to organize the data within tables and column families, and how to build partitions. to do this we transform the natural language query in a domain to look for the best querying language;
- Derived values: on the basis of entities and queries we might need to create additional attributes, made by derived data, that can be even built dynamically.

Remember that a NoSQL database is mostly used for only one application - this is especially true because we do not use normalized data.

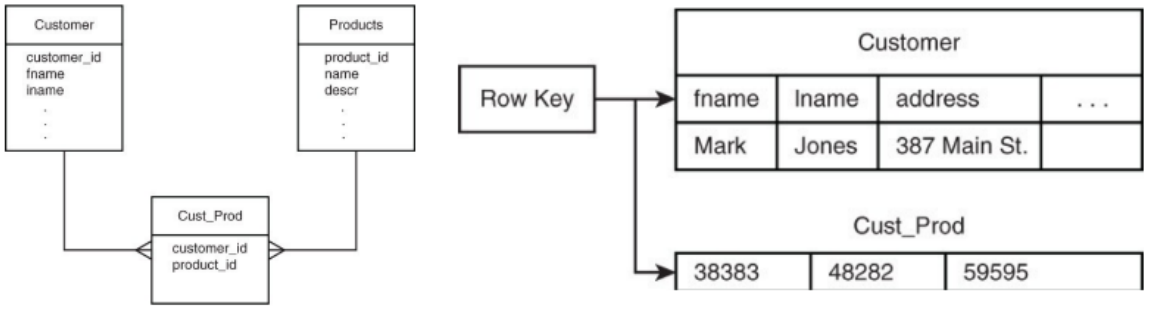
There aren't very many differences with relational databases - even the querying language can be the same!
There are though, significant ones, like:

- Column databases are implemented as sparse and multidimensional maps
- Columns can vary between rows
- Columns can be added dynamically - the only thing to define is the column family
- Joins are not used because data is denormalized
- It is suggested to use a separate keyspace for each application

Sparse and multidimensional maps are not really a problem. We do not need to allocate null values.

Let's talk about denormalization and how we can use columns and column families in an unconventional way.

Example: Many to many relationships



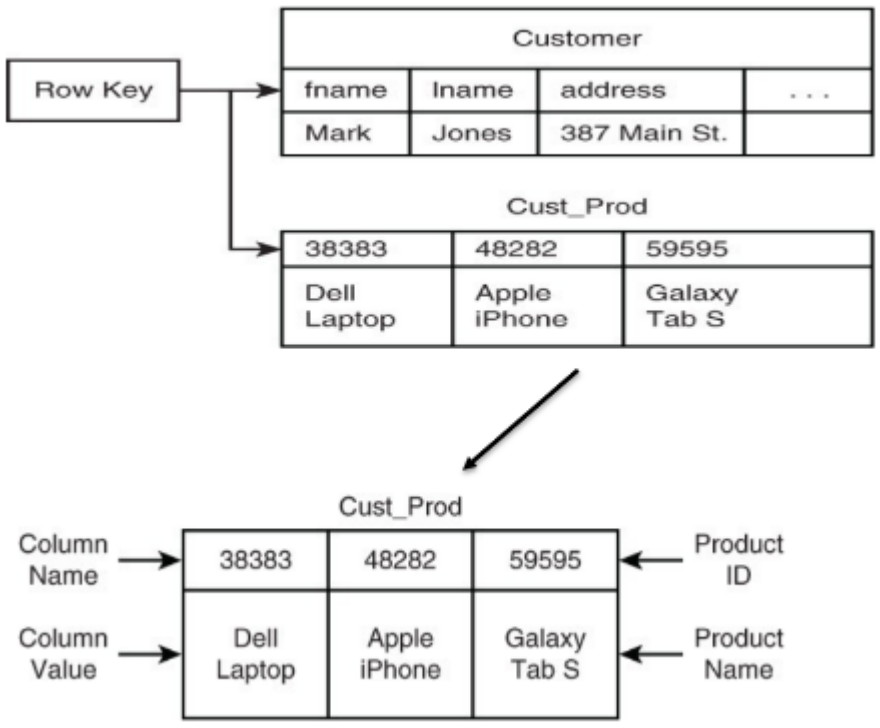
Relational DB

We use 3 tables, including a join table.

Column DB

In order to avoid join operations, each customer includes a ***set of column names*** that correspond to purchased products (Ids).

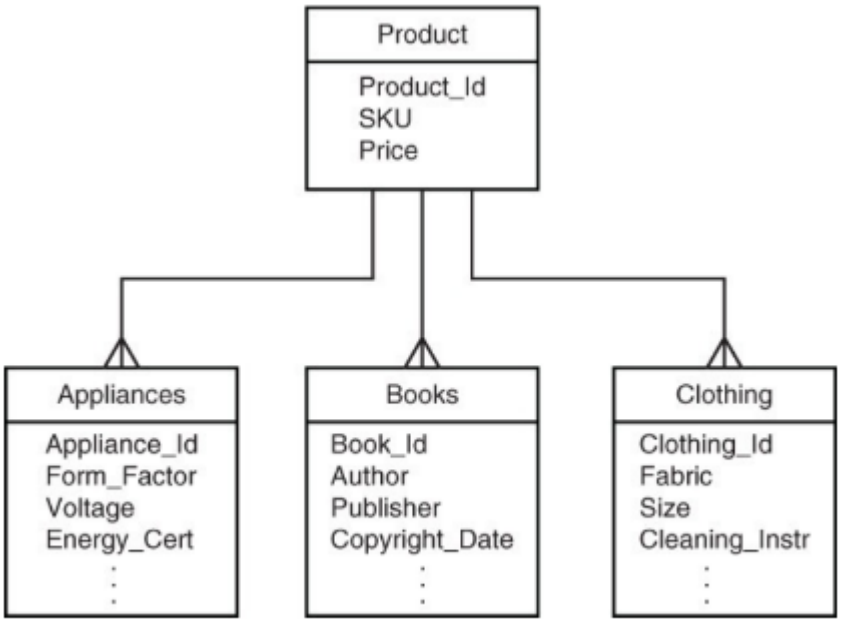
Let's suppose we have a customer that can buy many things at the store, and products that can be bought by several customers.
How can we use columnar databases to write and read quickly?
We need 2 tables, one for the customers, one for the products. Let's focus on the customer.
We can define a column family that takes all the attributes that describe the customer, with the anagraphics.
Then, we can define another column family, which is used as a list of the bought products.
This will represent the many-to-many relationship.
Whenever a new product is bought, a new column is created in the column family with the ID of the product as the name. So, we are storing a value in the name of the column. This allows us to count the number of products we bought and the IDs. If we need more info on the product, we can search for it. In this case, for another customer we will have different columns.



In the product tables, the same

thing will happen, but with the ID of the customers.
Actually, it can be more useful to exploit also the fact that we can specify a value for the column that we generated. For example, in the picture, we can put the description of the product as a value for the column with the ID of the product. It is a redundancy and a form of denormalization, of course, and we can do the same thing on the other table.
This way, if we want to quickly port the product bought by the customer, we can access only the customer table rather than retrieving the ID of the product and accessing another table.
Keeping a copy of the product name in the customer table will increase the amount of storage used but improve read performance.

We can imagine that Column DB is the OLAP part of a more complicated system and each night we run an ETL process for building the columnar database to be used for analytics.



Now, let's see a typical situation of having an entity, which is a product, that can be specialized in more ways.

All these items have some common attributes.

In column databases we can represent a specific instance of one of these three entities as a row in a product table of a columnar db.
So, we can have one big table and four column families to simulate the specialization.
For one appliance, we will have the row id, the product columns and the appliance columns.
THis is not a problem because we do not store null values, and we are sure that the info will be stored close on the disk and on the servers.

THe limitation is that in this case if we add a new product we must define a new table, a new column family, it's like changing the scheme in a relational database. One solution can be adding a "other products" family, but it is not a very good solution because we might mix together a lot of information.

There is a phenomenon called Hotspotting. We experience it whenever one of the servers of our clusters, or a group of them is in charge of managing most of the wead and write operations. There are some overloaded and some underloaded servers. This can be avoided, but how?
One solution is to hash the row keys in order to distribute the load evenly.

Usually these databases are filled using information from other systems, and the data during the ETL process is produced in an incremental way. This means that if we are reading information on other systems and writing them in ours, when we write we do so in a sequential way - mostly all on the same server. The others will stop.
We can add a random string as a prefix to the sequential value.
These strategies would eliminate the effects of the lexicographic order of the source file on the data load process.
Another way to avoid hotspotting is using parallel loading of the data, by using different threads.

Let's talk about versioning: some frameworks allow us to specify the timestamp of when the data was introduced. We might have different values with different timestamps regarding the same entity: this allows us to rollback if needed.
The DBMSs, by design, have settings to manage the versions - like for example minimum and maximum number of data versions. The number of versions to maintain depends on application requirements.
Versioning is useful to roll back changes, but many versions occupy a lot of memory. There is a tradeoff.

There are some DBMSs for columnar databases that allow us to store any object as a value, even a json file. Is it actually useful to do so? If we put an entire json as a value we do not exploit the features of this kind of database.
If we put the whole json in the database as a value, MAYBE the column database is not the best choice of architecture. If we do not actually need to do this, by doing this we are not exploiting all the features of the database like accessing the attributes easily and indexing them, utilizing column families…

Few words on indexing columnar databases.
An index is a specific data structure in a DBMS that allows the database engine to retrieve data faster than it otherwise would.
In columnar databases there is a primary index built automatically on the row key. This means that keys are organized in order to quickly retrieve the row by the DBMS.
Usually the user can create secondary indexes on one or more column values.

There is an issue: most of the DBMSs allow us to set indexes. There are some that do not have this feature, and we can switch off secondary indexes and implement them application side.
Usually a common sense rule states that "if we need secondary indexes on column values and the column family database system provides automatically managed secondary indexes, then we should use them."
Most database management systems will use the most selective index first. For example, in this query

```
SELECT fname, lname FROM customers WHERE state = 'OR' AND lname = 'Smith'
```

If we have the indexes for the states and for the surnames, we will first select the one index for which we have less records. The system will recognize the most selective one automatically.

Another advantage for using secondary indexes: let's imagine having built the application following the original requirements. Maybe some queries might be added later. Having indexes can speed up queries that appear later - building indexes lets us improve performance without having to manipulate the entire database again.

There are some situations, though, where it is better to avoid using automatically managed indexes.

- A situation where we have few values that can be assumed by an attribute, and evenly distributed, like a yes or no situation. we will probably have to have the same number of scans, while having additional delay problems caused by the data structure;
- A situation where we have too many distinct values, like a list of email addresses, because we will have an entry in the index for basically every entry in the database;
- A situation where we have too few values associated with a specific column, and it is not worth it to create and manage another data structure.

In these cases, we might want to build our own index with our own organization, from the side of the application. We can build additional tables to store the data we would like to access via the index.
We can have two ways to create and manage ad hoc secondary indexes, where the programmer is responsible.

When i make a write operation i have to update the table, the records and the data structure, the table of the index.
The first way I can do this is by updating the database, then the index, then return the control to the application. In this case we generate an additional extra time for concluding the write operation, but we are sure everything is updated.
The second way I can do this is by using batch updates, that is to say that we update the tables, we return the control to the application and there will be a batch job to update the index table. This introduces an eventual consistency, but it is acceptable in some cases.

The last thing regards atomicity of the write operation: it is guaranteed at the level of the column family. This architecture cannot be used for transactions, groups of rows do not have atomicity guaranteed.

Graph Databases: introduction

keep in mind that using a graph database is only a right decision when the data you want to manage and represent is in the form of a graph. This database is good for networks.

When we talk about graphs, we consider Vertices and Edges.
A vertex represents an entity - more than the entity, in the graph we look at the instances of the entities. We can give properties to the vertices, like attributes to the entities. The entities also have an ID, and one vertex is similar to a record in a table.
We create a graph whenever we define relationships between vertices. These are relationships between two instances of two entities.

We may specify properties linked to the relationships, and a weight is a commonly used property, which represents some value about the strength of the relationship like cost, distance etc.

Directed edges have a direction, whereas undirected edges do not have a direction.
The presence or not of a direction in edges depends on specific applications.

A path through a graph is a set of vertices along with the edges between those vertices.
Paths allow us to capture information about the relationships of vertices in a graph.
There may be multiple paths between two vertices. In this case (highways networks, for instance), often the problem of finding the best (the shortest, the cheapest, etc.) path is taken into consideration.

A loop represents a relation that an entity has with itself.
Proteins, for example, may interact with proteins of the same type.
The presence of a loop may not have sense in some specific domains, such as family tree graphs.

Obviously, all these choices depend on the nature of the data we have to model.

Some typical operations on graphs are Union and Intersection. The union is the operation of putting together two graphs by building a bigger graph containing all the vertices and edges of the first two, whilst the intersection is the graph built with only the vertices and the edges in common between the two graphs.

We will see that in graph databases there are three phases of defining a query. The first one is the definition of requirements in words, then we have the definition of the query in terms of typical graph actions, and then the translation in query language.

Graph traversal is the process of visiting all vertices in a graph. The purpose of this is usually to either set or read some property value in a graph.
The traversal of the graph is often carried out in a particular way, such as minimizing the cost of the path.

Very important because, depending on the type of query, we might need to traverse the whole graph.
For example, if I have a graph with students and courses and I have a query that asks for the average of all the students and to return the student with the highest average, I would have to visit the whole graph. This is not a good query and maybe I could use a column database instead.

isomorphism is another interesting concept.
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.

For example, Facebook and Linkedin: we can look at both networks to see if there are some isomorphisms to see if people work together and are also friends.

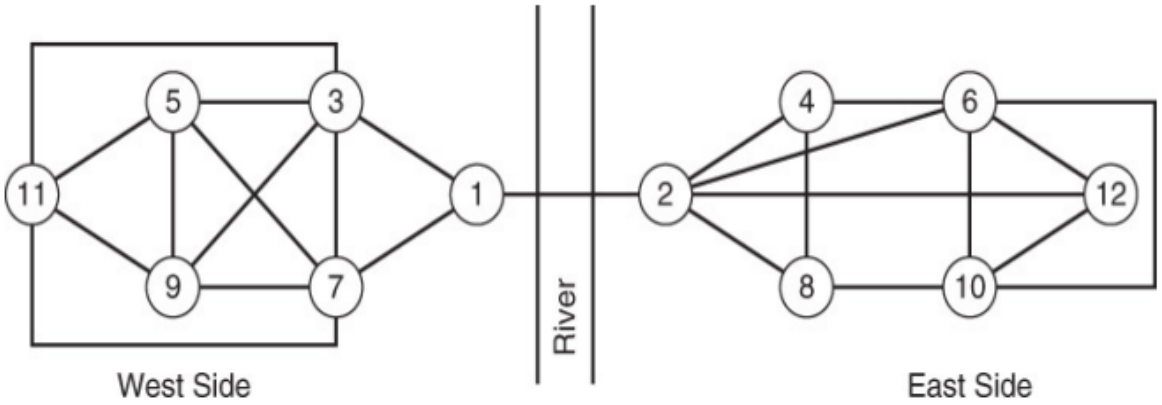
Some important measures:

- 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.
Example: to find the largest subset of people in a social network that know each other.
The higher the order and the size of the graph, the harder it will be to solve the query!

In order to estimate how useful and important a vertex is in a graph, we use the degree.
The degree is the number of edges linked to a vertex.
Closeness is the measure of how far the vertex is from all others in the graph. This value 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.
The concept of closeness is useful if we want to evaluate the speed of information spreading in a network, starting from a specific node.

Another concept is betweenness. It is a measure of how much of a bottleneck a given vertex is.



In this example, entities 1 and 2 if removed disconnect the graph.

Flow networks are direct graphs with capacitated edges: each vertex has some incoming and outcoming edges, and in each vertex the sum of capacities of incoming edges cannot be greater than the sum of the capacities of outcoming edges (apart from source and sink nodes). One example might be water pipes.

Another kind of graph is the bipartite graph. In this graph we have two distinct types of entities, and all instances of one entity are on one side, all the others are on the other side. Each vertex is then only connected with vertices of the other set - an example could be of students and teachers.

A multigraph is a graph in which we may have different kinds of edges between vertices. Each type of edge can be equipped with different sets of properties. One example could be the means of transport between cities.

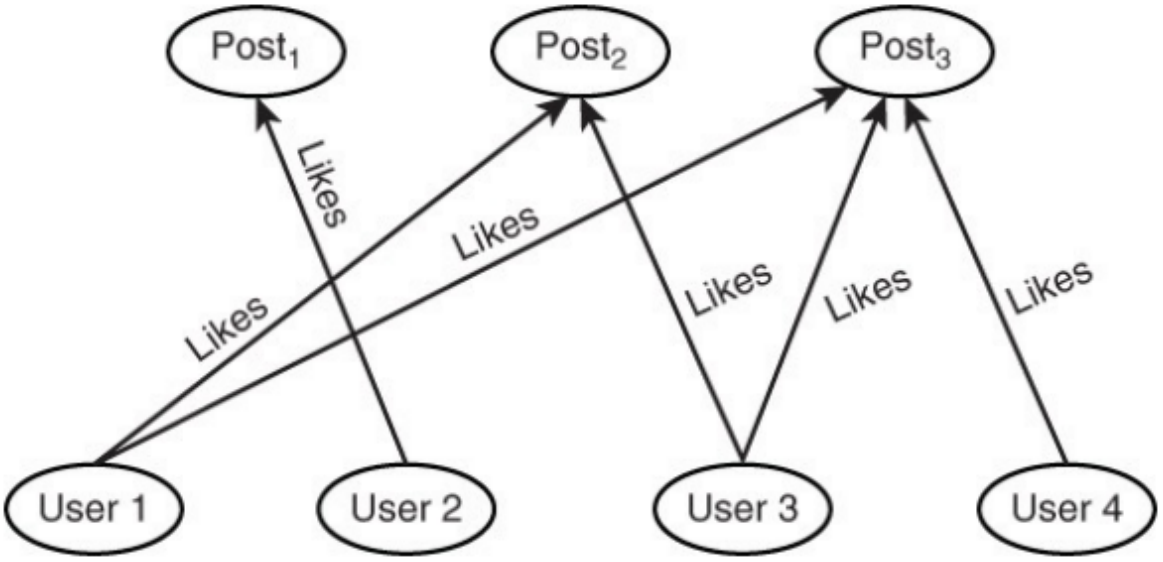
Now, we get more in depth with databases.
We will use the graph data structure to store data. This graph will be persistent on the disk and managed by a DBMS which usually has also a specific query language.
We have vertices, edges and properties for both of them.

If we want to model an highway for example, we can have an oriented graph in which we can specify how to move from one city to another.
Still, it is just a possibility. If we decide to only use one edge without direction between two cities, the edge will model the approximate distance and road traffic in both directions. If we want to provide more details we can use two directioned edges. It all depends on the level of detail we need our application to have. As always, it depends. One general rule, though, is to avoid putting too much information on the graph.

Another example of using the graph in epidemiology: our entities are people and an edge is a meeting between two people. Which properties can we use? Speaking of the vertices, apart from the name of the person, we could have some type of label, like “infected”, “immune/vaccinated”, “not infected”, “not vaccinated”, “infected in the past” etc.
We can put the probability of infection on the edges, and by analyzing the entire graph we can look at the probability of being infected for one person.

Another example of a graph is a tree to show part-of relation, like a hierarchy where the upper vertex is the parent vertex which can have multiple children vertices.

Here is an example on how we can model social media:



The objective of the graph database is to store explicit relations between entities. Of course we can do the same thing in relational database, where connections are represented as common attributes and join operations are used to find connections or links.

The problem is that dealing with huge amount of data requires a very computationally expensive join operation. Instead of making joins, we can exploit the fact that graph

databases have pointers inside of their vertices, and going from one entity to the other is easier and faster.
If we want to retrieve all the posts liked by a user we identify the user and we have all the pointers!

Different types of relations between entities may be handled exploiting multiple types of edges, and in each one of them we can specify different values of edge properties. For example, if we have vertices as cities we can set each edge as a means of transportation and have multiple between two vertices. We just have to specify the different properties for each one of them.

Graph Databases: design

Now we will talk about understanding how to exploit the graph architecture to build a database suitable for certain types of applications - generally networking organizations of the data.

When dealing with NoSQL databases, it is not easy to decide which solution should be adopted. In general, database designers should have in mind the main kinds of queries that the application will perform on the data.
Graph databases are suitable for problem domains easily described in terms of entities and relations.

In general, from the requirements and the queries you get to the architecture.

Typical queries for graph databases:

- How many hops (that is, edges) does it take to get from vertex A to vertex B? (middle-level query linguistically speaking, this could be a “how many cities do i have to pass while going from X to Y?)
- How many edges between vertex A and vertex B have a cost that is less than 100?
- How many edges are linked to vertex A? (this is the degree of the vertex)
- What is the centrality measure of vertex B?
- Is vertex C a bottleneck; that is, if vertex C is removed, which parts of the graph become disconnected?

These are all queries related to a network application.
There is less emphasis on selecting particular properties or attributed, and less emphasis on aggregating values across a group of entities (we do not really look for average values nor sums).
Moreover, the previous queries are fairly abstract (computer science/math domain).
Designer needs to translate queries from the problem domain - like natural language - to the domain of experts working on graphs.

Let’s make an example: the DB for handling movies, actors and directors.
Let suppose to offer the following services for users:

1. Find all the actors that have ever appeared as a co-star in a movie starring a specific actor (for example: Keanu Reeves).
2. Find all movies and actors directed by a specific director (for example Andy Wachowski).
3. Find all movies in which specific actor has starred in, together with all co-stars and directors

Now, we can solve this with a relational database. We will need four tables (Movies, Directors, Actors, People) - still, SQL lacks the syntax to perform the queries as of above, it does not traverse graphs in an efficient way, especially when the depth is not bounded.
Performance is affected heavily by traversing the graph, and each level of traversal adds delay to the response time of the queries.

In a graph database, we only need a Film and a Person entity, and two different types of edge (directed, and acted).
The query language too - Cypher in Neo4J - makes it way easier to navigate the database and to write queries. The nice thing is that the link between entities exploits pointers at low


```
neo4j-sh (?)$ MATCH (kenau:Person {name:"Keanu Reeves"})
                    -[:ACTED_IN]->(movie)<-[:ACTED_IN]-(coStar)
              RETURN coStar.name;
```

level, making it fast to traverse the graph. This is called Free Adjacency Index property.

Another example, we want to design a social network for NoSQL database developers.
The main use cases may be:

1. Join and leave the site
2. Follow the postings of other developers
3. Post questions for others with expertise in a particular area
4. Suggest new connections with other developers based on shared interests

5. Rank members according to their number of connections, posts, and answers

The main actor is the database developer - which is a user of the social network, of course.

Once we have these requirements at high level, we must identify entities with properties.

Developers:

- Name
- Location
- NoSQL databases used
- Years of experience
- Areas of interest

Posts:

- Date created
- Topic keywords
- Post type (for example, question, tip, news)
- Title
- Body of post

If we do not really need to put all these data on the graph. The graph might only be useful to represent the connections, and the whole data could as well be in another database. This makes the graph lighter. The information that is needed should always be put on the graph to avoid having to query the other database every time, but if a piece of information is not needed it should not stay on the graph.

Once we identify the entities we need to analyze the relations - the ones that characterize one instance and another. We do not really delve into the realm of what is one to one, many to many etc.

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.

So, now we analyze the details of the relations.

What about developer x developer? It is not very difficult to understand that we can create the following relationship for developers. This is important because we can see that developers can create posts, and the application might show friends' posts to the users. I might also be interested in the friends of friends' posts. We are not obliged in defining the number of hops that will be traveled, if not at the level of application code.

We can define even more relationships.

Between developers and posts we can decide on different things - two directed edges, or one non directed one. The directions could have different meanings.

Another interesting relation here is the post to post one. One post could be an answer to another post.

In our project, if you want to use a graph database, you will have queries in domain specific language. We need to translate it in a graph centric query.

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?

Guidelines to follow for designing this kind of database are:
Make sure that at least part of your data model can be seen as a graph. After that, identify the queries, make the use case diagram and the uml class one. Then, identify the entities and the relations between them.
Then, map the domain specific queries into more abstract ones, and implement graph queries - and use graph algorithms.
Pay much attention to the dimension of your graph. The algorithms on graphs are not linear, mostly. The bigger the graph, the more computationally expensive it will be to perform queries.

If available, define and use indexes for improving retrieval time - only use them when it is necessary, not for queries which should not be done on a graph database in the first place.

Use appropriate types of edges, undirected and without properties if possible - if the relations are symmetrical, you do not need directed edges, and pay much attention to the data type of the properties.
Another important thing is to watch for cycles when traversing graphs. Use appropriate data structure for keeping track of visited nodes.

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

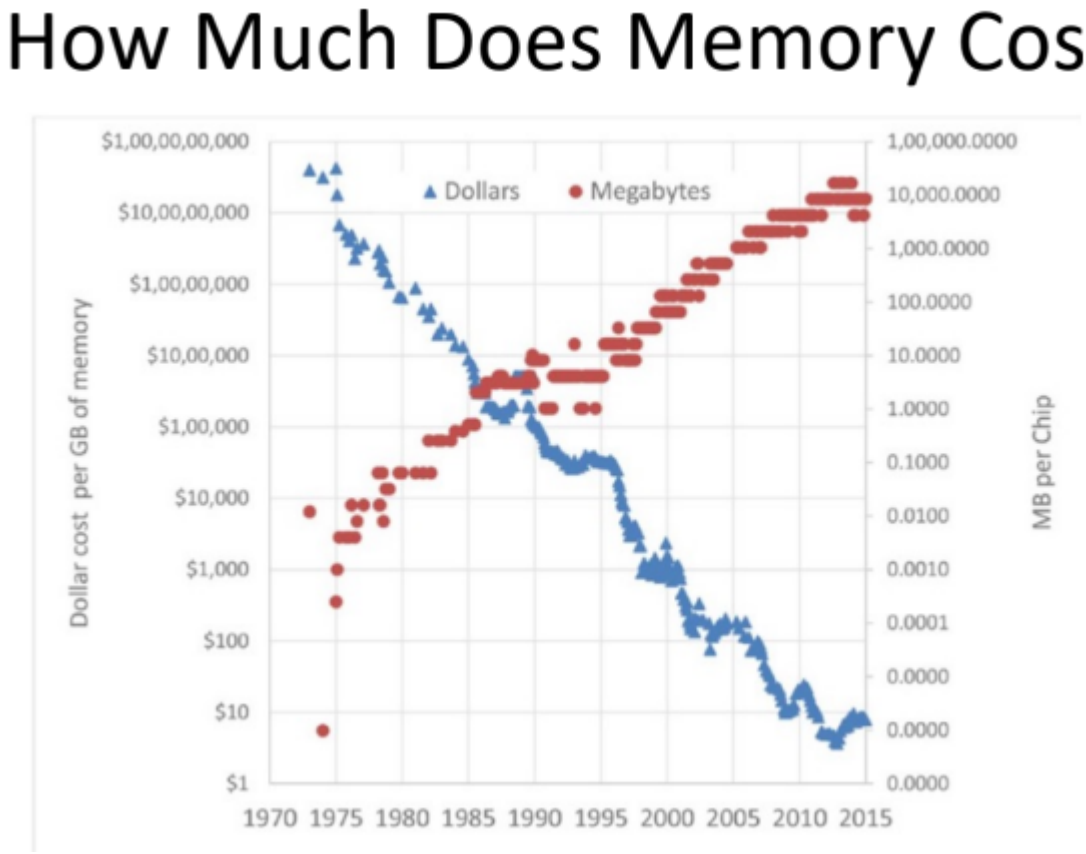
Some time ago, Graph Databases were not scalable - and the community version still is not scalable, mostly. They can only be vertically scaled. Thus, the developer must be careful with the growth of the

number of nodes, edges, users, properties.

In Memory Databases

If our application does not need to handle huge amounts of data, nor does it need to scale - either vertically or horizontally - we might want to adopt another solution.
In-memory databases may be suitable for applications whose handled data may fit all in the memory of a single server.

We can ensure availability, still, by residing in the memory capacity of a cluster, if replication is needed.



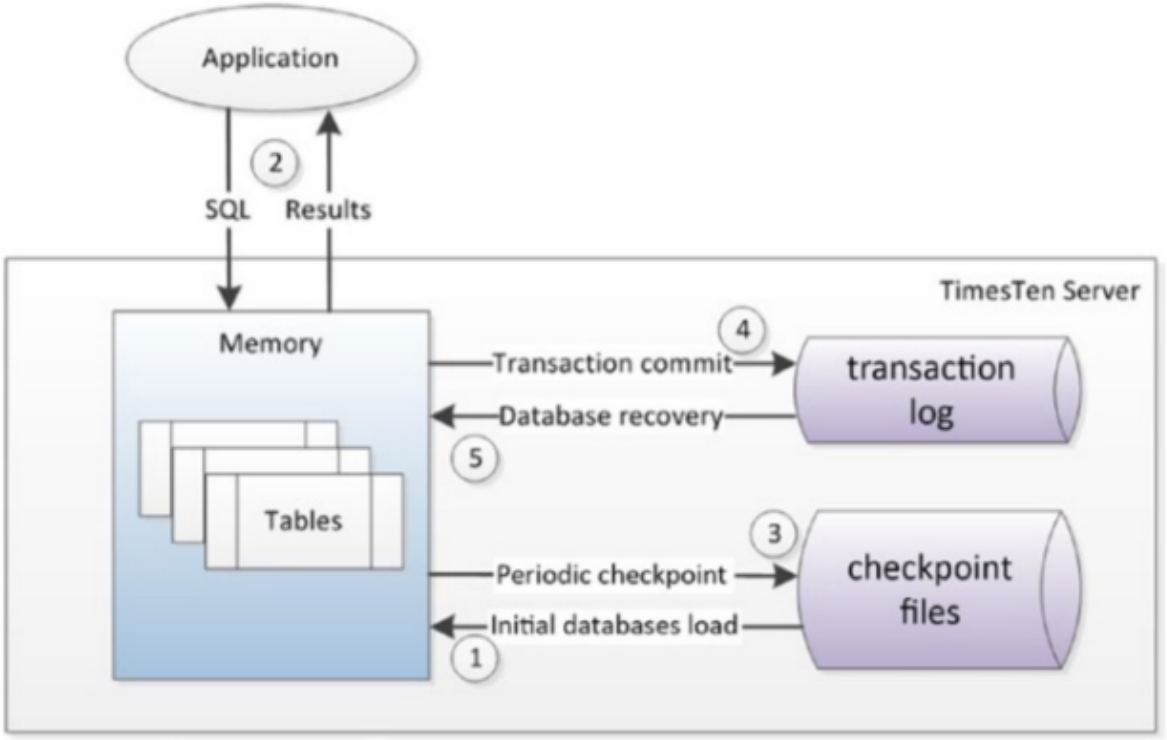
Memory is not so expensive anymore.
The features of in memory databases include a very fast access to the data, by avoiding the bottleneck of the I/O transfers. These databases are suited for applications that do not need to write continuously to a persistent medium. We can also avoid having cache memory, since everything is already in memory. There has to be an alternative persistence model, since we want to avoid data loss in the event of a power failure.

Solutions for data persistency might be:

1. writing complete database images, called snapshots, to disk files, like level db (key value);
2. writing out all the operation records and transactions to a transaction log or journal or in general any append only disk file - a bit like a commit log
3. replicating data to other members of a cluster.

Here are some examples of architecture.
Keep in mind that during the exam I could ask how to approach the problem of persistence in memory databases. The answer we just gave is good in general, but if I ask about the architecture I want to know the specific solution.

TimesTen, built by Oracle, implements a fairly similar SQL-based relational model. This means that it was born even before NoSQL.
All data is memory resident.
Persistence is achieved by writing periodic snapshots of memory to disk, and a transaction log following transaction commits. When we have a transaction, we make a commit. We can return the control to the application directly after the commit.
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; this will translate into a performance loss, though.



This is the architecture. In memory we have the relational organization of the table. The application uses SQL and retrieves quickly the results from the memory. On the disk there are checkpoint files where we make snapshots with a certain frequency, and the append only transaction log. The transaction log contains information on timestamp, transaction and value for each transaction. We may decide to update the transaction log after each transaction, or in an asynchronous way, after giving control back to the application.
If we do this last thing, we do not ensure ACID transactions. If we want to be sure to have durability of each transaction, after it is done we must update the log. Of course, we will lose in terms of performance.
The periodic checkpoint is done at a distance that is decided by the administrator.
The transaction log is used to recover from failures.
The checkpoint is used whenever we start back the application.

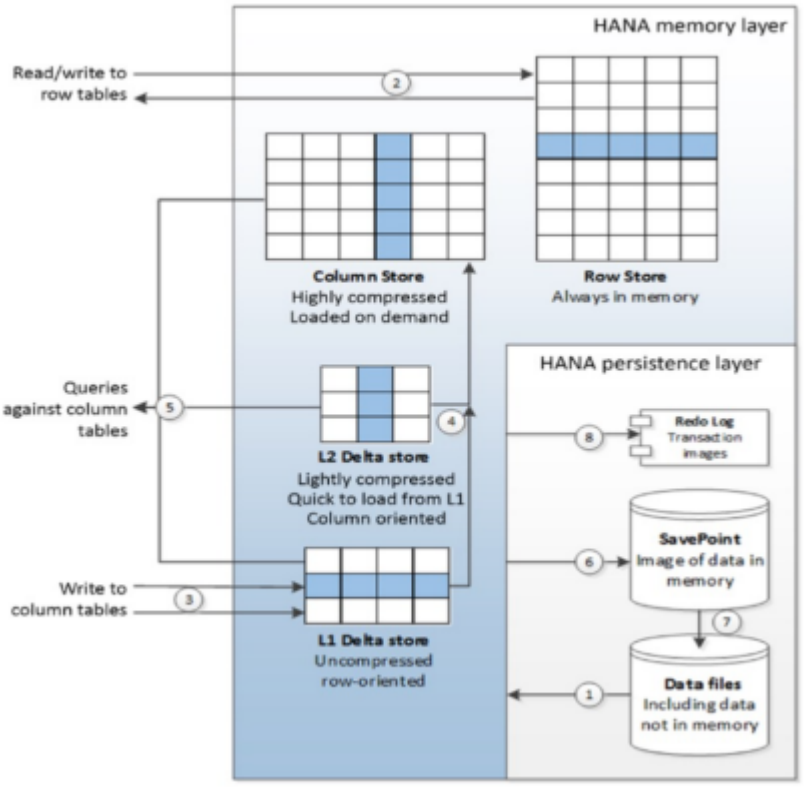
Redis is an in memory key-value database. The values that can be considered are lists of strings, or hash maps. These are used for creating queues of requests. The server can access them and process them.
Only primary key lookups are supported, and there are no secondary indexing mechanisms allowed.

One point of strength is the fact that Redis may exploit a kind of “virtual memory” mechanism for storing an amount of data larger than the available memory of the server - the older keys are swapped out to the disk and recovered if needed, with a mechanism of paging that costs a bit in terms of performance.
Keys can be stored in the disk if they are not used frequently.

This is the Redis architecture. We may decide to have replicas, then we have the snapshot of the database that we can make at some frequency, than we have an AOF file in which we can log the transaction, then we have the swapping space. So, if we want to improve our performance we can decide to make replicas to reduce the level of persistence on the memory.
Obviously, in case of write operation there has to be a sort of eventual consistency implemented, to handle replicas.

HANA DB is an SAP product, and it is a relational database which combines an in memory technology with a columnar storage solution and row organization of the database.
We may have some ad-hoc hardware solutions and configurations for working with this architecture.

Usually we have classical row tables used for OLTP operations, and stored in memory; columnar tables are used for OLAP mostly, and they are loaded in memory on demand and otherwise stay on the disk. Sometimes the columnar tables can be interested in OLTP operations - to write records in the columns. So, the database may load some columns in memory from the beginning.

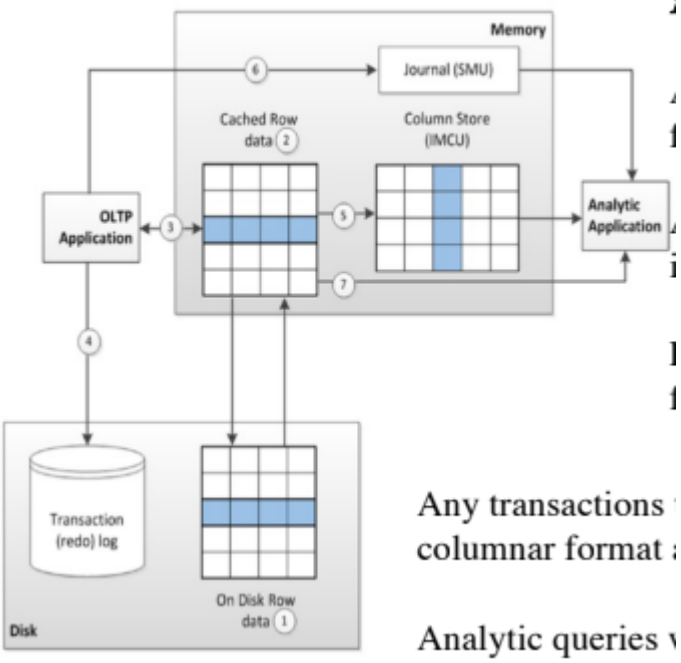


HANA has a main row oriented relational database, always in memory while running, and with snapshots. From the row store we make write and read operations. It's just queries and no analytics. Then, the system might be interested in some writes in the columnar part. The database designer will decide where each data goes, and after that the application manages that. So, when we have write operations towards the columnar part, these are first cached in memory by an uncompressed row oriented database. To use them in analytics, these are then updated in a delta store and then in another columnar database. The information is transformed in a sort of fast ETL towards a more compressed and better organized columnar store - when we make queries, we ask to the delta stores and to the column store in order to avoid missing information (the transformation of the data might still be in progress).
It is a very complex system.

The last one is the Oracle 12C.

Data is maintained in disk files, but cached in memory; an OLTP application primarily reads and writes from and on the memory.

Any committed transactions are written immediately to the transaction log on the disk.



Any transactions
columnar format

Analytic queries

Row data is then loaded into a columnar representation that can be used in order to make analytics.
Any transactions that are committed once the data is loaded into the columnar format are recorded in a journal, which will be consulted by any analytic query to determine if they need to read updated data from the row store or to rebuild the columnar structure.
The way the two twin memories are managed depends on the caching strategy.

Guidelines for selecting a database

In order to design and develop a specific software application, we need to perform a set of steps:

1. Get the requirements from the customer
2. Define functional and non functional requirements
3. Define the use case scenario, better if you do it with UML diagrams
4. Identify the main data structures, the main procedures and the relationships among them
5. Refine the previous step, project workflow and including db design
6. Implement and test
7. Deploy

In order to do this the right way, the Agile Approach can let us revise all the mistakes and things that we can do better, and even change requirements and design.

The role of the engineer, in this situation, is to identify the most suitable software and hardware architecture, the most efficient programming languages and development environments and the most fitting database management system.

Everything must, and should, be driven by the requirements, the engineer's experience, and common sense.

One question that arises after the whole course: are relational databases still useful?
The answer is: yes. They protect data integrity, reduce the risk of anomalies, have a strong theoretical base and ensure immediate consistency and ACID transaction support. They will continue existing and supporting OTPL applications - moreover, there are decades of experience and a lot of best practices and design principles that make it a still valid solution for many applications.

Still, in the era of today, performance and availability are more important than consistency and ACID transactions in many of the newer applications and data types.
We must measure ourselves with Big Data, Internet Of Things, Large scale web applications, and Mobile applications. In these new entities, relational databases are not useful anymore, they are too strict, too rigid.

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 DynamoDB

- Document databases, such as MongoDB and CouchDB
- Column family databases, such as Cassandra and HBase
- Graph databases, such as Neo4j and Titan

The choice between these is guided by requirements. The functional ones are related to the type of queries that describe how the data will be used, influencing the nature of relations among entities and the needs of flexibility in data models.

There are also 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 do not allow too much flexibility and the possibility of making queries on different attributes. They are preferable when:
- The application has to handle frequent but small read and write operations
- The data models are very simple
- Simple queries are needed, with no complex queries and filtering on different fields - if you write too much code it's the wrong architecture

Some examples of applications that may exploit these databases:
- Caching data from relational databases to improve performance
- tracking transient attributes in a web app, like a shopping cart
- storing configuration and user data information for mobile applications
- storing large objects such as images and audio files

Document databases have flexibility as their main feature, along with high performance capabilities and easiness of use - embedded documents allow us to use a few collections of documents that store together attributes that are frequently accessed together (denormalization). They are suitable for applications that need:
- a variety of attributed to be stored, both in terms of number and type
- to store an elaborate large amount of data
- to make complex queries on different types of attributes
- to use indexing, which are very powerful
- to make advanced filtering on attributes

Some examples of applications that may exploit these databases:
- Back end support for websites with large amounts 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 that benefit from denormalization by embedding structures within structures

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, we should avoid these databases. They are suitable for application that:
- Require the ability to read and write frequently to the database
- Are geographically distributed over multiple data centers
- Can tolerate some short term inconsistency in replicas
- Manage dynamic fields

- Deal with huge amounts of data, such as hundreds of terabytes

Some examples of applications that may exploit these databases:

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

Graph databases are suitable for applications that:

- Need to represent their domain as networks of connected entities (two orders in an e-commerce are not connected to each other, neither do game player's configurations with each other; proteins interacting with other proteins and employees in a firm, yes they do!)
- Handle instances of entities that have relations to other instances of entities
- Require to rapidly traverse paths between entities.

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

When choosing solutions for data storage and management, we are not obliged to use just one type of database management system.
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 include 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.
Another: OLTP handled a Relational Database (for ACID transactions), OLAP in charge of MongoDB for fast analytics.

EXAM QUESTIONS

Situations in which it is best to use a key Value Database. Why are they the fastest? Why are they used in websites' shopping carts?
They are the fastest because they are in memory.

Which architecture is better for OOP? Why is a schema less architecture better? Which advantages does it give?
Document databases are the best, they are schemaless and the structure of the data inside it recalls the one of a class. It lets us handle hereditariety better.

Which are the principal characteristics of Big Data? Why can't we use classic DB architectures like relational databases to handle them?

On massive applications, why do we need an architecture like Cassandra and not MongoDB? How can we choose?

Why would I prefer a serverless vs a distributed (mongoDB) approach? What is the difference between a read and a write in Cassandra vs. in MongoDB?
The distributed nature of cassandra can modulate better a big system

What are indexes? Are there indexes in relational DBs?
Yes, you can make them everywhere

What are the main drawbacks of indexes? Which queries are faster?

Network partition: which are the problems? How can we handle them?

What is eventual consistency? What is causal consistency?

If I have many replicas, which are the issues? Are they happening in reading or writing?

Is it ok to choose only one architecture in a project? When I design the database of an application, which are the issues of using two architectures?
Strict or eventual consistency?
Can you give an example of an application with two architectures?

What are the differences between a relational and a column database?
What are the differences with the third normal form?
What are the differences between the many-to-many relationships?

Index free adjacency?

A graph database makes the graph structure explicit. In a graph database, each vertex serves as a mini index of its adjacent elements because it has the pointers that point to them. as the graph grows in size, the cost of a step to an adjacent element stays the same.

how to approach the problem of persistency in in memory databases?