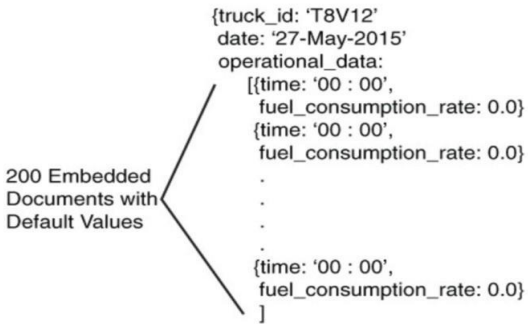
UNIVERSITÀ DI PISA

LARGE-SCALE AND MULTI-STRUCTURED DATABASES

Class Notes 2023/2024

Last Update: Thursday, 22 February 2024

Sommario



Last Update: Thursday, 22 February 2024

Read me

This paper has been written by students who have attended the course of Prof. Pietro Ducange during the academic year 2023 – 2024.

The entire paper is written in English (*maybe you can find some words in Italian, in that case simply translate it*).

These notes are based on the lecture and on the material that the teacher gave us.

We did our best to make these notes more accurate as possible.
However, we do not guarantee the perfect accuracy of the content (*in case of any doubt, ask to the teacher*).

For this reason, we do not recommend using these notes without following the professor's course.
Use this resource as a supplement to, and not as a replacement for the course.

We would like to thank ZIOne for the support and help to maintain this note.

We wish you a good study!

Prerequisites

You SHOULD have knowledge about:

- Java Client-Server Applications and some API's.
    - If you don't have those, go read about the course "Programmazione Avanzata".
- Relational Database.
    - If you don't have those, go read about the course "Basi di Dati".
- Software Engineering and UML diagrams.
    - If you don't have those, go read about the course "Ingegneria del Software".
- Unix-Based Operating Systems.

Information for programming in Java

IDE → Eclipse

Connection to MySQL → JDBC(*using Maven for handling dependencies*).

Last Update: Thursday, 22 February 2024

Introduction

Big Data Era

Big data is an actual situation that we are experiencing every day.

ACID Transaction

ACID (Atomicity, Consistency, Integrity, and Durability) is an acronym that describes the fundamental properties of a relational database. We are moving from **ACID** to **BASE** (Basic Available, Soft State and Eventually Consistent).

Key-Values Databases

Is something like the vocabulary, the value is identified by a key, a key can be an array of more complicated things like a string. A record is identified by a key.

Document Databases

More complicated architecture, attributes and properties are described by a JSON file.

Columnar Databases

Usually we manage a table, which describes a specific entity. One record is stored in a block of the disc. Is easy to have direct access to one specific entry. But if i want to calculate a statistic from a specific attribute i have to get all records from the disc, then extract the attribute that i want and finally calculate the statistic. Is very time-consuming. In the column database we have similar attributes in the same block of the disc, so retrieving all salaries from the database is very easy, but it is difficult to retrieve an entire record.

Graph Databases

We'll study this type of database.

Big Data

It is a "*fashion-buzzword*"…everyone talks about it, but what is it really?
Big data is not a technology, it's something.
The word represents a big amount of data received from different sources (*internet*) that continuously generate and send data at different speeds (*often very high speed*).

The 5 V's

Figure 1- The 5 V's

Volume
Files are sended continuously on the internet, but you should be aware that even a simple file of a few Megabytes contains millions of bytes.

Veracity
If I collect some data at time T, at time T+1 are they still valid or they must be updated/deleted?

Different Sources

In different sources we can find data about the same thing, but they could be inconsistent, they could be different. This is also a problem of security of the sources. What do I have to do if I receive incorrect data or that comes from a not secure source?

Velocity
Every source has a different speed that can depend on many conditions (*network, load of work…*).

Variety
Sources can be everything connected to the internet that produce, collect, and send data like security cameras, laboratories, transactions, mobile devices and so on. Every source can produce many files of different format and size.

Value

Raw data do not have any value; their value (*knowledge*) must be extracted. Every data has value, and nobody gives data for free. Data has to be interpreted, alone they have no meaning, so we have to give them a meaning.

How to manage Big Data

A big data scientist should have knowledge of:

Analytics and Mining

How to retrieve meaningful data.

Database

How to manage data.

Computation Infrastructure

Where to elaborate data.

Storage Infrastructure

Where to store data.

Security and Privacy

How to protect data and so the clients.

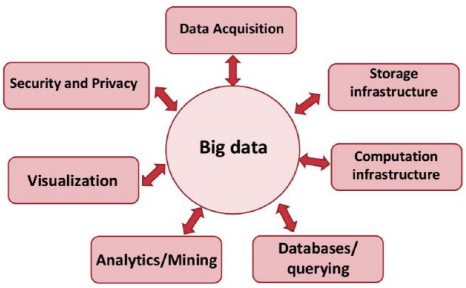


Figure 2- Type of operations that concern Big Data

The main Big Data computing issues

Classical data elaboration algorithms, such as the ones for analytics and data mining, cannot directly apply 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.

Vertical Scaling

If you need more, then upgrade your machines to increase the elaboration capacity. But, with this technique I can issue:

- Single point of failure.
- Data are physically centralised (*too much data in a single storage*).
- Cost of the single machine.

Horizontal Scaling

If you need more, then buy more machines and distribute the incoming jobs.

- Data are physically distributed (*but logically centralised*), they are spread in different databases.
- We need more complexity to manage distributed data (*because I want that logically data appear like they are stored in a unique place*).

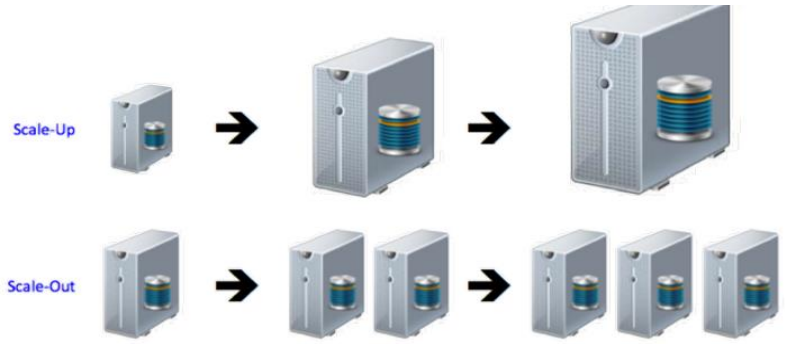


Figure 3- Vertical and Horizontal Scaling

Distributed Systems

Which is the Objective?

To apply an operation to all data. But one machine cannot process or store all data. So, we need a distributed system (*horizontal scaling*).

What is distributed system and Why we need it?

A distributed system is any environment where multiple computers or devices are working on a variety of tasks and components, all spread across a network. Components within distributed systems split up the work, coordinating efforts to complete a given job more efficiently than if only a single device ran it.

Properties of a distributed system

- Data is distributed in a cluster of nodes.
  - We still have to know where data physically are.
- It does not matter which node executes the operation.
  - It does not matter if the same job is run multiple times in different nodes (*due to failures or straggler nodes*).
- We look for an abstraction of the complexity behind distributed systems.
  - We want that all the architecture appears like a unique computer.

Data Locality

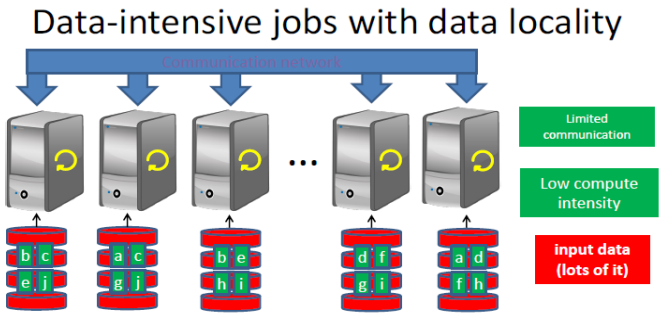


Figure 4- Data Locality

The process of moving computation to the node where that data resides, instead of vice versa. This technique helps to minimise network congestion and improve computation throughput. We store data on disks of the nodes that perform computations on that data. Note that we should avoid data transfers between machines as much as possible; because moving data from disk to memory (*and vice versa*) is time expensive.

MapReduce

It's a whole new paradigm introduced by Google in 2004. MapReduce is a framework for distributed computation on a huge quantity of data using a large cluster of computers.

"Moving computation is cheaper than moving computation and data at the same time".

Data is distributed among nodes (*distributed file system*), operations are distributed among nodes. It follows the "*divide & conquer*" idea:

Divide (map)

Divide the partition dataset into smaller, independent chunks to be processed in parallel.

Conquer (reduce)

combine, merge, or otherwise aggregate the results from the previous step.

The MapReduce Workflow

MapReduce consist in 6 steps:

Input Reading

We receive and unstructured data input from a source.

Splitting

We divide the input dataset in N smaller and independent chunks to be processed in N parallel nodes.

Map

Mapping is a partial elaboration that every node does on their own. Each node (*of the N nodes that we speak about in the splitting*) will process one of the N chunks. Every node will produce a partial result.

Shuffle

We group the partial results. The grouping operation is based on the type of partial result.

Reduce

In the reduce step we group the `N` partial results into a unique result, which is part of the final output of the computation.

Result

We merge all the parts, and we finally obtain the final output.

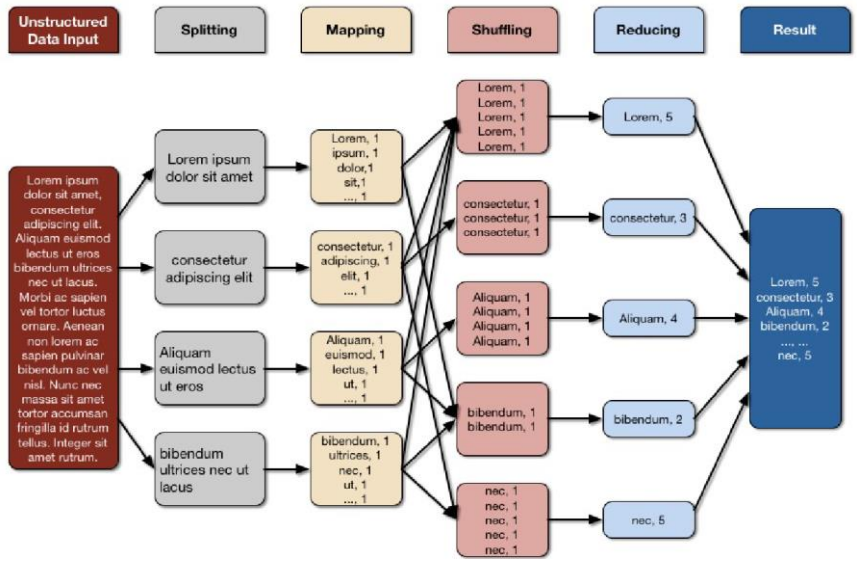


Figure 5- MapReduce workflow

MapReduce - practical explanation

In this example, the computation consists in calculating the number of occurrences of letters in an input text.

1. The initial text is divided into 3 smaller blocks.
2. Each block goes in input at one mapper; each mapper produces a partial result:
   - "*For each letter we have the number of that letter in the text*".
3. Then partial results go into a reducer that aggregates them to produce a part of the output:
   - "*Calculate the addition of the occurrences of a specific letter*".
4. All the results produced by reducer are merged in the final output.

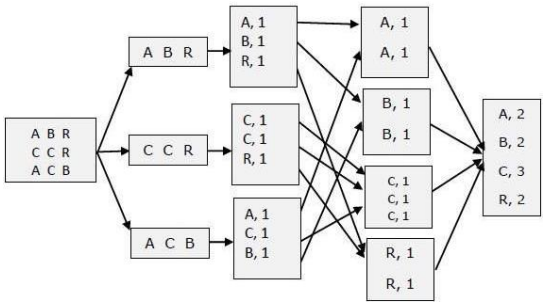


Figure 6- Example of MapReduce

MapReduce - formal explanation

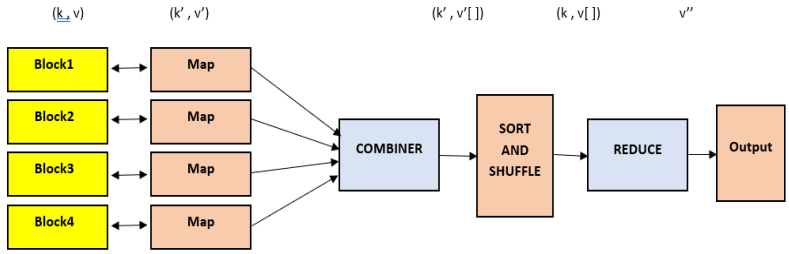


Figure 7- A Forma explanation of the MapReduce workflow

1. Initially data are divided in chunks.
   - A chunk is something like `<K , V>`.
2. Every mapper receives one chunk and produces a partial result.
   - A partial result is something like `<K' , V'>`.
3. Then the partial results are sorted and shuffled.
   - In this step they are regrouped using the key `K'`, so the result of the shuffling is the list of map results regrouped by key.
   - The result of the shuffling is something like `<K' , V[]>`.
4. Then there is the reduce operation that produces the final output by merging the result of the shuffling.

Hadoop

It is an open-source framework (*written in Java*) that we can use for running applications that manage big data. Hadoop uses MapReduce for perform distributed computations. Hadoop consists of several modules, each one implementing a specific functionality: one of those modules is called `HadoopDistributedFileSystem(HDFS)` and it implements a distributed file system (*thanks to that module I can use the distributed storage like a unique big storage*).

Figure 8 – Hadoop's logo.

Properties of Hadoop

Horizontal Scaling and Data Locality

Hadoop can scale to clusters with thousands of computing nodes, each node will store a part of the data in the system.

Fault Tolerance

Hadoop is robust to faults because it has data replication (*a partition of the data can be replicated in more than one node*).

Designed for low-cost hardware

Designed for managing big files using low-cost hardware.

Efficient for read operations

Efficient for read and append operations (*random updates are to consider rare*).

Limits of Hadoop

Hadoop is efficient for one-shot (*read and append*) operation, but for operations that include more iteration of Map and Reduce it's not the best choice. Hadoop doesn't have a good memory computation to do that, in fact for each MapReduce operation we have to do an access to the disk for loading data from disc to primary memory. A write operation needs an access to the disk, so it is really slow and represents a bottleneck.

Spark

Appreciated for its enhanced flexibility and efficiency: Spark uses a master/slave architecture i.e., one central coordinator and many distributed workers. Here, the central coordinator is called "*driver*", and the workers are called "*executors*". Each executor is a separate process.

Figure 9 – Spark's logo.

Properties of Spark

Flexibility on the distributed model

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

In-memory cluster computing

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 until the last operation, at the end of the map reducing operation the memory is flushed.

Independent processes

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.

How Spark works

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.

The cluster Manager does the resource allocating work.

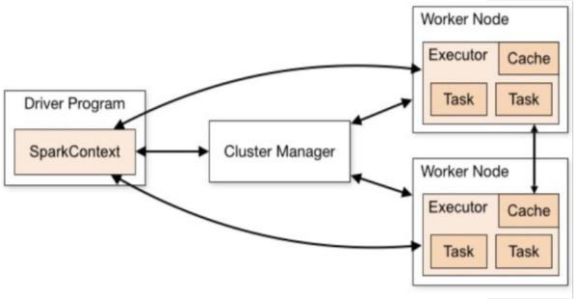


Figure 10- Simplified Spark's workflow.

The First Database Revolution – The DBMS

The DBMS represented a new layer that allowed the separation between the database handling logic (*functional integrity, join etc*) and the application itself. As a result, programmer overhead was minimised, and the performance and integrity of data access routines were ensured.
The two Main Features of a DBMS are:

- The definition of a scheme for the data to store in the database.
- Give an access path for navigating among data records.

The Navigational Models and their issues

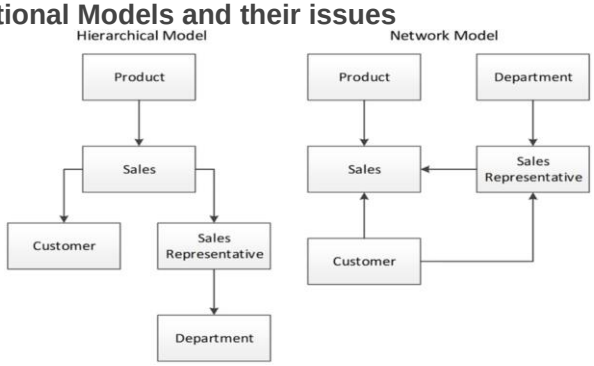


Figure 11- The classical DBMS Model.

Lack of Flexibility

They *ran* exclusively on the *mainframe computer* systems of the day (*largely IBM mainframes*) Only *queries* that could be *anticipated* during the initial design phase were possible.

Complexity

It was *extremely difficult* to add new data elements to an existing system. It is not clear the logical and physical division of data.

Time-Hungry for application developing

CRUD (Create, Read, Update, Delete) operations oriented: complex analytic queries required hard coding: Indeed, programmers had to write methods for reading, modifying files, tables and in general for interacting with stored data.

The Second Database Revolution

The business demands for *analytic-style reports* were growing rapidly. Existing databases had some problems:

- Were too hard to use.
- Lacked a theoretical foundation.
- Mixed logical and physical implementations.

Relational Theory

Now we will see a recap of the traditional relational theory.

Tuples

A tuple is an unordered set of attribute values. In a database system, a tuple corresponds to a row, and an attribute to a column value of the tuple.

Relations

A relation is a collection of distinct tuples and corresponds to tables in relational database implementations.

Constraints of the Relational Theory

Constraints enforce consistency of the database. *Key constraints* are used to identify tuples and relationships between tuples.

Operation and the Relational Theory

Operations are made 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. You must set your queries in advance.

The Third Normal Form

Non-key attributes must be dependent on “the key, the whole key, and nothing but the key”.

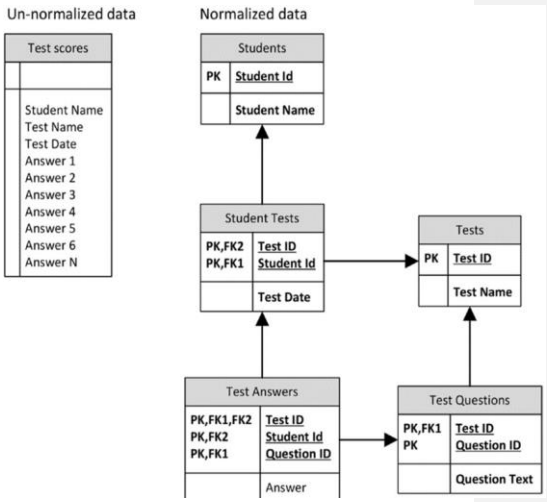


Figure 12- Normalisation of the data

Transactions

A transaction is a concurrent data change request on a database system. We need rules to ensure consistency and integrity of data.

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 ACID Transaction

Consider the following transaction T consisting of T1 and T2:
Transfer of 100 from account X to account Y.
If the transaction *fails* after completion of T1 but before completion of T2 (*after write(X) but before write(Y)*), then the amount has been deducted from X but not added to Y.
This result leaves the database in an *inconsistent* state.

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

Figure 13- Example of ACID transaction

Object-Oriented DBMS - OODBMS

Object-oriented Programming vs RDBMS

Object-oriented (OO) developers were frustrated by the mismatch between the object-oriented representations of their data within their programs and the relational representation within the database. In the program a whole entity was represented with only one class, but in the database the entity was represented with several tuples in different tables.

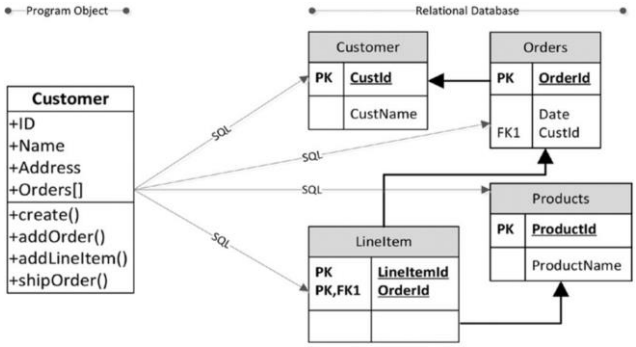


Figure 14 - How an object is converted in the normalisation form.

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

The encapsulation is important: the class also has public and private attributes and methods: people can only access to the field programmers want to, These characteristics are lost in a relational model. 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.

OODBMS - A Solution for OO Programmer Frustration

A new DBMS model, namely Object-Oriented Database Management System (OODBMS). An OODBMS permits you to store objects without normalisation.

- Applications would be able to load and store their object very easily.
- The implementation resembles the "Navigation" models.

However, OODBMS systems had completely failed to gain market share, But Object-Relational Mapping (ORM) frameworks (Hibernate for example) has been introduced to alleviate the pain of OO programmers.

Massive web-scale applications - MWSAs

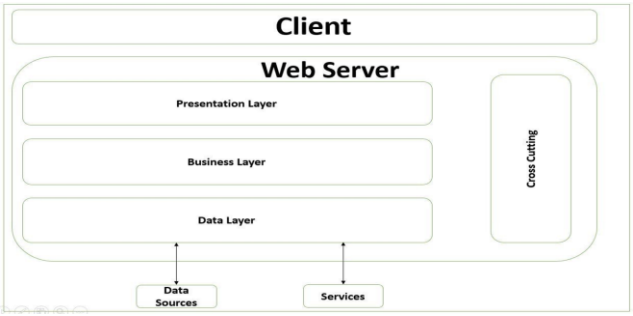


Figure 15- simplified diagram of a MWSA

Big companies started to put pressure on the traditional paradigm because they need:

- Large volume of read and write operations.
- Low latency responses time.
- High Availability.
    - The system should continue to work correctly and should always provide feedback.

The problems of the relational database

The scaling

The performance of RDMSs may be improved by *upgrading* CPUs, *adding* memory and *faster* storage devices, but the vertical scaling has some problems:

- C*ostly*, in terms of hardware and maintenance.
- There are *limitations* on the number of CPUs and memory dimension supported by a server.

The horizontal scaling has its problems too: It's very hard to manage a distributed relational database.

- *Performance issues* related to some operations that involve tables stored on different nodes.
- *ACID transactions* are hard to implement on multiple servers.

Modify the schema

And finally, in a RDBMS it's *hard* to modify the structure of existing data (*i.e., adding an attribute to an object stored in a table*).

The Needs of modern companies

Scalability

We have a spike (*picco*) of traffic on a website, then we add new servers (*scaling-out or "temporary horizontal scaling"*) that we shut down when the spike ends; but it is hard to adopt this solution with a RDBMS, because we need to manage all the replicas and manage all the updates. We need fast and cheap horizontal scaling.

Flexibility

E-commerce has millions of products, all different, so an object can have some columns that other objects don't need to have. We want a scheme-less model, a model that doesn't give a limit on type and attributes that I want to use. Every object has its model, we shouldn't have to assume the category of an object. In a RDBMS it's very hard to change the scheme once you have data in the database.

Availability

The system should always be available for all users all around the world (*companies lose money if their services are offline*). So, the database must be distributed to reduce the latency at the minimum, so, if a server crashes or has too much work, there should be another server ready to take its job even though the prestation go down. The RDBMS isn't the correct choice because managing tables on a distributed database is very time consuming.

Low Costs

RDBMS have licence costs that often depend on the number of users. A solution could be to use an open-source DBMS, but an enterprise then has to change the entire infrastructure.

NoSQL Databases

"NoSQL" stands for "Not Only SQL".

NoSQL considers a set of data models and software. Most NoSQL solutions ensure: *Scalability*, *Availability*, *Flexibility* and often they are *open source.* ACID may not be supported on a NoSQL database. NoSQL rejects the constraint of the relational model, including the strict consistency and schemas. But there exists also NoSQL databases which respect ACID properties and the features of a relational database. SQL Cloud-based solutions are still largely used (*Amazon and Google*).

DBMS and Distributed Systems

General tasks of a DBMS

Databases must allow users to store and retrieve data. To follow this aim, three tasks must be in charge to a DBMS:

- Store data persistently.
- Maintain data consistency.
- Ensure data availability (*important non-functional requirement*).

Properties of distributed systems

Most of the recent NoSQL DBMSs can be deployed and used on distributed systems, namely on multiple servers rather than a single machine, that can communicate with each other with a network.

Scalability, flexibility, cost control and availability

If you want to consume less power, you can switch off some of the servers and turn on when you need to.

Easy horizontal scaling

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)

Easy to improve

Allow the implementation of fault tolerance strategies and spikes management.

“Theoretically right”

Accomplish the motivations that led to the third database revolution!

I can’t have everything

To balance the requirements of data consistency and system availability. Because if we have replicas, we need to ensure the consistency, therefore I must update the information’s of all replicas (*time consuming*).

Nodes isolation

To protect themselves from network failures that may leave some nodes isolated.

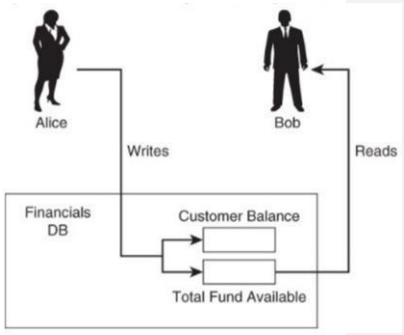
Data persistency

Data must be stored in a way that is not lost when the database server is shut down.

Data consistency

In this figure we can see two users, Alice, and Bob, we are in a financial database. The customer balance represents the money that Bob can send to the company when he buys a product from it and the Total Fund Available represents the money that company can use for making business.

1. Bob buys something from the company.
2. Alice has to decrease his Balance and increase the company's Balance
3. In the meanwhile, If Bob tries to read the Total Fund Available before Alice completes the transfer, he will read a wrong (*not updated*) value.

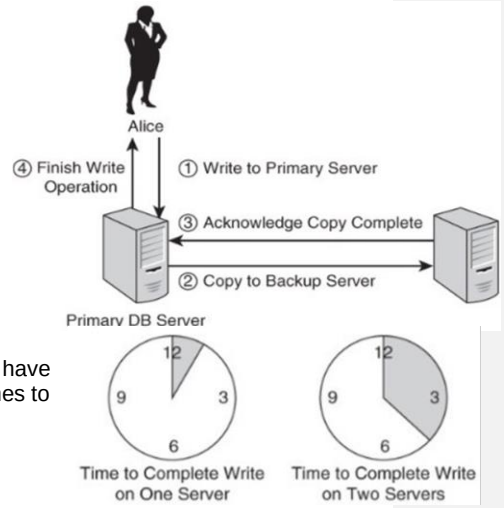


Data availability

The data availability refers to the fact that data is always available to be read or to be modified. 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.

Two-Phase Commit

Ensures a high data availability: all the transactions on the primary server are replicated on the backup server, and the final commit is sended to the user only when both servers have done the transaction.
If the primary server goes down, the Backup server takes its place, and the DBMS continues to offer its services to the users. The two-phase commit is a whole transaction itself, so it should be ACID! You can have consistent data, but transactions will require longer times to execute than if you did not have those requirements.



ACID Properties

Jim Gray definition of ACID:

"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)"

An ACID transaction must 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.

"TOO ACID!"

There are some applications in which it is not acceptable waiting 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.
Other applications need consistency first, like every application that concerns the financial world. The system must be always available (*during and ACID transaction the system is not available*), otherwise I will lose money! So, the response time should be minimum.

Consistency

NoSQL databases often implement eventual consistency. There might be a period where copies of data have different values, but eventually ("*prima o poi*") all copies will have the same value.

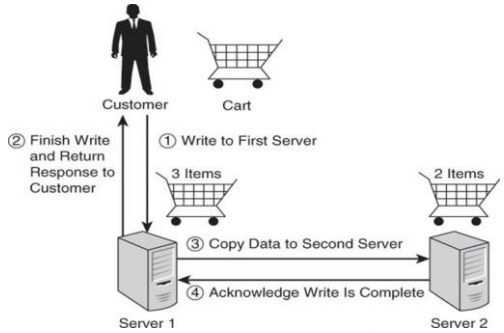


Figure 16 - Maintain a backup copy of the cart.

A word about availability

Amazon: "even an "insignificant" tenth of a second (0.1s) in the response time of the site generates a reduction in sales estimated at around 1%".

Available but not Consistent

In the e-commerce scenario the cart may have a backup copy that may be outdated. But the system would be "more available" because the data would still be available if the primary server failed. But the replica on the backup server would be inconsistent with data on the primary server if the primary server failed prior to updating the backup server. But the system is still available, and the user can still insert products in the cart. If I lost some product from the cart, the user would simply put them in the cart again.

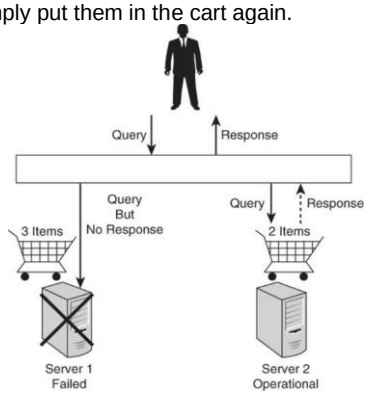


Figure 17- Available but not consistent.

Consistent but not Available

Let's suppose we use the two-phase commit for have the maximum consistency level on the replicas. When dealing with two-phase commits we risk that the most recent data not being

available for a brief period, because the two-phase commit is very time-consuming. While the two-phase commit is executing, every other query to the data is blocked.

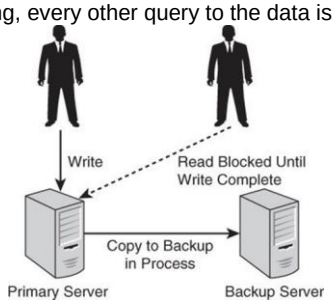


Figure 18- Consistent but not available until the two-phase commit is going on.

Network partitioning

A network partition is a division of a computer network into relatively independent subnets, either by design, to optimise them separately, or due to the failure of network devices. Distributed software must be designed to be partition-tolerant, that is, even after the network is partitioned, it still works correctly. A system has two choices:

1. Show each user a different view of the data (*availability but not consistency*) but keep itself available.
2. Shut down one of the partitions and disconnect one of the users (*consistency but not availability*).

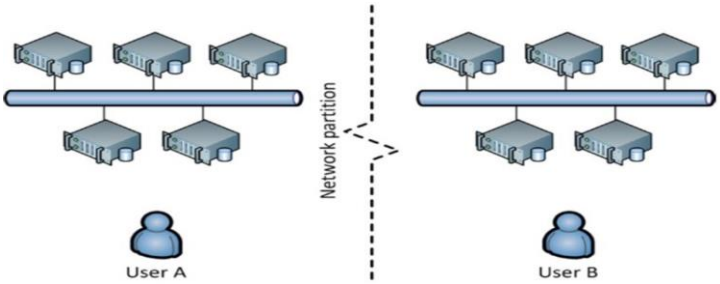


Figure 19 - Network partitioning.

Properties of a network partition

- Each partition is independent, so is one of them fails the other can still work.
- Each partition has its nodes and replicas so it may happen that an entire partition has inconsistent data.
- If a partition fails, some terminals may not be available.
- Some node may can't communicate with some other nodes due to the separation of the subnets.

The CAP theorem - "*Brewer's theorem*"

Distributed Databases cannot ensure at the same time:

Consistency (C)

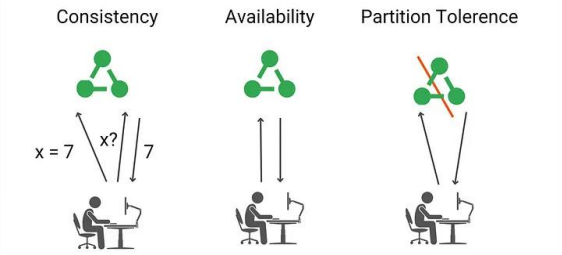
Data is the same across the cluster, so you can read/ write from/to any node and always get the same result.

Availability (A)

The ability to access the cluster even if a node in the cluster goes down.

Partition Tolerance (P)

The system's ability to continue functioning despite the presence of network partitions *("network partitioning", or simply "partitioning"*) or communication breakdowns between nodes.



The CAP theorem thesis and the CAP triangle

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

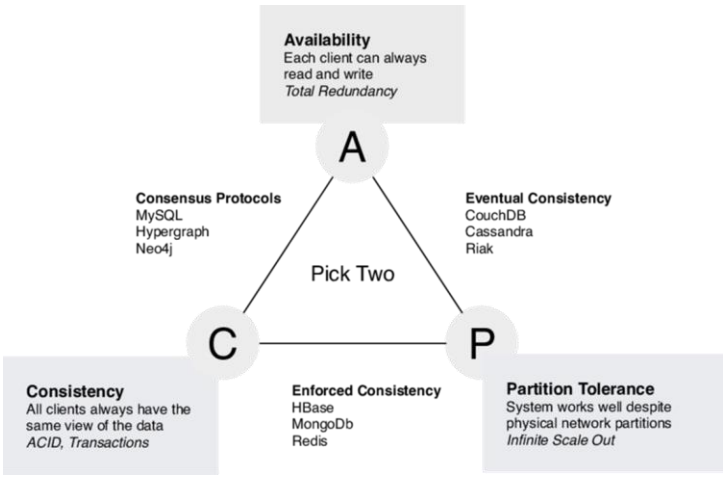


Figure 20 - The CAP triangle.

CA Solutions
We want consistency and availability, but the system can be affected by partitioning. We don't have network partitioning; therefore, all nodes are always in contact, but when a partition fail occurs, the system blocks. *Use cases*: Banking and Finance application, system which must have transactions e.g., connected to RDBMS. Total consistency can affect performance (*latency*) and scalability.

AP Solutions

System is still available under a partition fail or partitioning, but some of the data returned may be inaccurate. Indeed, this system state will also accept writes that can be processed later when the partition is resolved. Availability is also a compelling option when the system needs to continue to function despite external errors. *Use cases*: shopping carts, News publishing CMS, etc.

CP Solutions

I don't care about availability (*i can accept that the system is working for someone and for someone else don't*), but thanks to partition tolerance in another terminal the system is still working. Suitable if long response times are acceptable (*Bank ATMs*). They are usually based on distributed and replicated relational or NoSQL systems supporting CP. In most of the recent NoSQL frameworks the availability level can be set up with different parameters.

Choose wisely! Carefully read the requirement analysis.

And remember:

ALWAYS CONTEXTUALISE YOUR CHOICES

BASE properties of NoSQL databases

BA - "Basically Available"

Reading and writing operations are available as much as possible *(using all nodes of a database cluster*) but an operation might not be consistent (*the write might not persist after conflicts are reconciled, and the read might not get the latest write*).

S - "Soft state"

This property refers to the fact that the state of the database can change over time, even without any explicit user intervention. This can happen due to the effects of background processes, updates to data, and other factors. The database should be designed to handle this change gracefully and ensure that it does not lead to data corruption or loss.

E - "Eventually consistent"

At some point (*I don't know precisely when*) in the future, data in all nodes will converge to the consistent state.

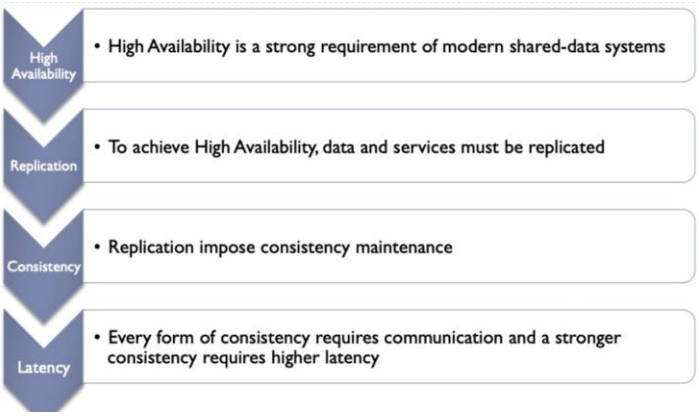


Figure 21 - Base application needs

Consistency Management

I have to deal with the update of all replicas, so i must implement a set of functions in charge to maintain the consistency of replicas. More requests I get, the more latency I will have, because maintaining consistency all over the replicas requires time. To minimize the latency, we must implement those functions properly (*avoid useless time-consuming operations*).

Strict Consistency

Strict consistency (*Strong Eventual Consistency*) is the strongest consistency model (*and the more time-consuming*). An update of a variable needs to be seen instantaneously by all users. All the replicas must have the same data (*and always the last version of them*) every time.

Eventual Consistency

The update will eventually be reflected in all nodes that store the data. In Eventual Consistency it's not sure that a user will always read the last version of the information, but eventually (*at some time in the future*), the user will read the last version of the value.

Read-Your-Writes Consistency

Once an item has been updated by a client C1, any attempt to read the very same record by C1 will always return the last version of the value. But the update of the same value on other replicas may not be instantaneous, so any other client that performs the same read operation immediately after the write operation of C1 could get an outdated value (*if the read operation is done in another server, otherwise the client will get the updated one*). This consistency is meant to reassure the user (*that made the write operation*) that his write operation is successful.

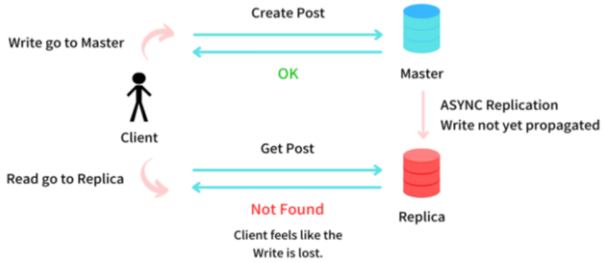


Figure 22 - Read-Your-Write consistency diagram,

Read-Your-Writes consistency example

1. Alice updates a customer's outstanding balance to $1,500.
2. The update is written to one server and the replication process begins updating other copies in the other servers.
3. During the replication process, Alice queries the customer's balance (*so Alice do not end the session previously opened*).
4. She is guaranteed to see $1,500 when the database supports read-your-writes consistency.

Session consistency

Ensures "*read-your-writes consistency*" but only during a session.

Figure 23 - Example of the Session consistency.

The animated graphic above illustrates the session consistency with musical notes. The "*West US 2 writer*" and the "*East US 2 reader*" are using the same session (*session A*) so they both read the same data at the same time. Whereas the "*Australia East*" region is using a different session (*session B*) so, it receives the updates later but in the same order.

What if the user changes 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. The user could not immediately see his updates, but this paradigm follows the eventual consistency, so *eventually he will be able to see his updates*.

Monotonic read consistency

If a process reads the value of `X`, any successive read operation on `X` by the same process will always return that same value or a more recent value. If another process makes the same query on the same location, it is not guaranteed that he will see the same version of the first user.

"The reads cannot go backwards"

Monotonic read consistency example

1. Alice updates the customer's outstanding balance to $1,500.
2. Alice updates the customer's outstanding balance to $2,500.
3. Bob queries the database and sees $2,500.
4. Bob queries the database and sees $2,500.

If Bob issues the query again, he will see the balance is $2,500 even if all the servers with copies of that customer's outstanding balance have not updated to the latest value.

Monotonic write consistency

If a process makes two write operations, they will be executed in the order he issued them, and every other process will see the two operations in the very same order. This paradigm can't be applied to updates made by different process on the same object.

Monotonic write consistency example

1. Alice reduces all customers' outstanding balances by 10%. (*sconto del 10%*)
2. Charlie has a $1,000 outstanding balance.
3. The reduction is applied: Charlie now has a $900 balance.
4. Alice continues to process orders.
5. Charlie ordered $1,100 worth of material.
6. Charlie's outstanding balance is now the sum of the previous outstanding balance:
    - $900 + $1100 = $2000.

Now consider what would happen if the database performed Alice's operations in a different order.

1. Alice reduces all customers' outstanding balances by 10%. (*sconto del 10%*)
2. Charlie has a $1,000 outstanding balance.
3. Charlie ordered $1,100 worth of material.
    - Charlie's balance becomes $1000 + $1100 = $2100.
4. The reduction is applied: Charlie's balance is set to $1890.

The balance of Charlie now is 1890 Instead of $2,000, so Alice lost $110. So, the order of the operations is very important.

Causal consistency

Is a particular type of eventual consistency.

Causal relationship

If an operation causally depends on a preceding operation, there is a causal relationship between the operations. This relationship could create side-effects that that the user must see. If an action A happens before an action B and there is a causal relationship between them, then the action A must always be executed before the action B on all replicas.

Why bother about these relations?

As we can see in the image below: between the post of Pietro and the post of Mike it is a casual relationship because Mike's post is a response to Pietro's post. If Pietro would not have written the post, then Mike would not have written the response. In the accepted view order, I maintain the casual relationship.

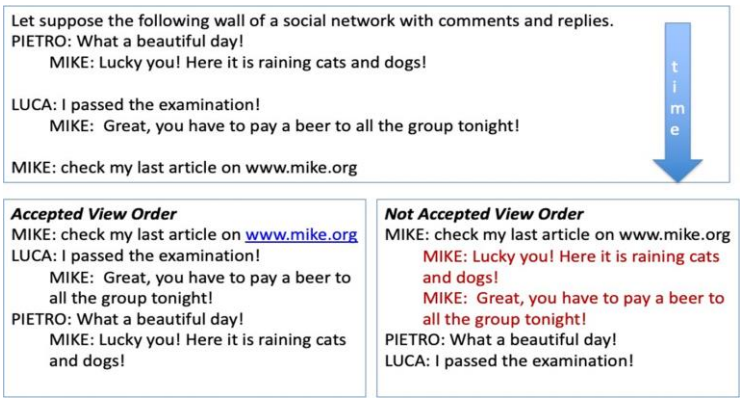


Figure 24- Example of causal relationship.

Key-Value databases

Classic array

It is a multidimensional data structure. An array is an *ordered* list of *values* and can be accessed with an index. Has some limitations:

- The index can only be an unsigned integer.
- All the values inside the array must be of the same type.

Associative array

Generalise the idea of the array. This type of array removes the limitations of the classic array:

- The *index* can be a *generic identifier.*
- The *values* can be of *different types*.

Dictionary

Is the data storage paradigm used in these types of databases. Dictionaries contain a collection of objects (*records*) which may have many different fields within them, each one containing data (*for example a JSON object*). These records can be retrieved using a key that uniquely identifies them.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Figure 25- Example of dictionary.

Keys and values can be anything from a simple object to complex compound objects. This type of structure permits to Key-value databases to be highly partitionable and horizontal scalable at levels that other types of databases cannot.

Namespace and buckets

The namespace is a first level container, and it is used to organize data. Meanwhile buckets are used to organize data within a namespace (*they usually contains highly correlated data*). The namespace is composed by one or more buckets. If you use more than one bucket, then the key is: `Final Key = < Bucket Name , Key >` so keys must be unique within the bucket.

```
Namespace: products
  Bucket: electronic
       Key: product123
      Value: {"name": "Phone1", "price": 599, "brand": "ABC"}
 Key: product124
      Value: {"name": "Phone2", "price": 699.99, "brand": "BBC"}
  Bucket: clothes
       Key: product456
      Value: {"name": "Shirt", "price": 79, "brand": "XYZ" }

Namespace: active_users
     Key: user123
     Value: {"name": "Mario", "surname": "Rossi", "score":12 }

Namespace: admins
     Key: admin1
     Value: {"name": "Luca", "surname ": "Bianchi"}
```

When should be chosen

When you need ease of storage

A key-value database requires a low quantity of memory

When you need a fast retrieval of data

If performance is more important than organising data into more complex data structures, such as tables or networks.

When you do not need "multi-table" features

Practically, when you do not need all the features of the relational models (*joining tables or running queries about multiple entities in the database*).

Essential features of a key-value database

Simplicity (*schema-less*)

Unlike a relational one, we do *not* have to *define* a database *schema* nor to define data *types* for each *attribute*. I only have to define the namespace and the buckets.
If we need to add a new attribute (*maybe because a certain entity needs, it)* we can simply modify the code of our program without the necessity of *modifying* the database.

Flexible

We can make mistakes assigning a wrong type of data to an attribute or use different types of data for a specific attribute.
The key-value database can also be used instead of a configuration file because it can be easily readed by an application.

Forgiving - No Typing for the Values

No typing Check: If I insert something (*like a telephone number*) with an inconsistent value (*like a string without numbers*), nobody will check this. This permits the system to be very fast.

```
String:'1232 NE River Ave, St. Louis, MO'

List of strings:('1232 NE River Ave', 'St. Louis',’MO')

JSON String:
{
      'Street:' : '1232 NE River Ave',
      'City' : 'St. Louis' ,
      'State' : 'MO'
}
```

For the key-value database are all the same (*strings*).

Caching System

The last updated version of the value is loaded in the primary memory and while the user program is doing something else using the cache the DBMS can write the recently updated values to disk. This trick permits the user to perform write operations at a very high speed and the disk will be eventually updated (*and eventually in a batch mode*) and it decrease drastically the number of accesses to the disk .

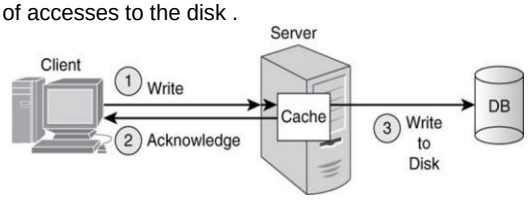


Figure 26 - Diagram of the caching system.

In fact, key-value databases stores are often used for in-memory data caching to accelerate application responses for read operations. Obviously will use a partial paging strategy to put data in the cache (*because i can’t put all the database in the cache*), but if the database is small enough, then it can be entirely loaded in the primary memory.
I can use this strategy on Relational databases too.

The Cache managing strategy

When the key-value database *uses all the RAM memory* allocated to it, the database will need to *free* some records to make space for the new ones.
Several algorithms exist for selecting the records to remove from the RAM and the *Least Recently Used* (LRU) is the most used one.

LRU

Data that has not been read or written as recently as other data can be removed from the cache.

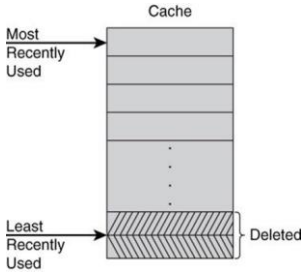


Figure 27- LRU algorithm.

Possible cases of the key-value database

E-commerce cart

Depending on the requirements: if the cart must be there after the log-out, then i must memorise the cart on the disk, if I don't need this, then i can keep the cart only in primary memory, so after some time LRU will delete the record (*but if the user immediately log in after the log out, the cart could be still there*).
Then when I make the checkout, the order will be stored persistently in another database (*relational or document*) and the cart emptied.

Scalability of a key-value database

Capability to add or remove servers from a cluster of servers as needed to accommodate the current load on the system. When dealing with *NoSQL databases*, employed in web and other large-scale applications, it is particularly important to correctly handle operations when the system is horizontally scaled.
I need the replication of the server. In the framework of *key-value databases*, two main approaches are adopted:

- *Master-Slave Replication.*
- *Masterless Replication.*

Master-Slave replication

The *master* accepts *write/read* requests, and it oversees the collecting of all the updates of the data for eventually be sent to all the slaves (*for update them too*).
The slaves may only respond to read requests, to do some load balance.
When a write operation is performed on the master node, it is eventually replicated to all the slave nodes to keep the data consistent across the entire system.
This type of architecture is particularly suitable for read-intensive applications, in other words for applications with a growing demand for read operations.

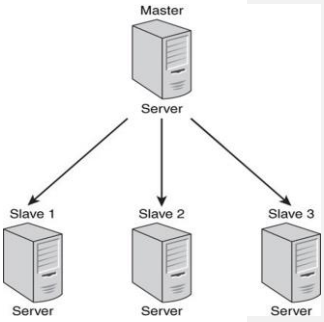


Figure 28 - Master-Slave replication.

Pros of the Master-Slave replication : Simplicity

Only the master communicates with the slave servers and the communication is "one-way". There is no need to coordinate write operations or resolve conflicts between multiple servers accepting write requests, the slave only receives updates from the master.

Cons of the Master-Slave replication : Single Point of Failure

If the master fails, the cluster cannot accept writes requests anymore, but it can still respond to any read request.

"***The Higher the Complexity, the Better the Availability***"

There is a strategy for respond to a master crash for keep the system available (*able to receive write requests*):

1. Periodically, each server (*master and slaves*) sends a message to another server in the cluster to check if it is still active (*if active, it will simply respond*).
2. If the master crashes, the slaves do not receive updates anymore (*or any other message from the master*).
3. If several slave servers *do not receive any response* from the master within some period, the slaves may determine that the master has failed.
4. The slaves initiate a protocol to *promote* one of the slaves as a *new master.*
5. A new master has been elected so the system continues to work normally.

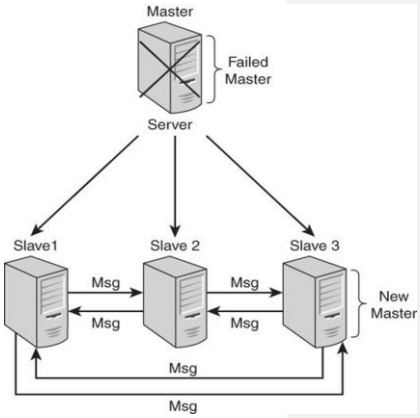


Figure 29- new master election.

"Morto il papa, se ne fa un altro"

Masterless Replication

The *master-slave replication* architecture *will suffer* an increasing number of write requests, because only the master accept writes and all the updates must be sent to the slaves. *Solution*: a replication model in a cluster of servers where all nodes may accept write requests.
I don't have a single point of failure because there is *not* a single server that has the *master copy* of updated data (*the most updated one*), and there is not a single server that can copy its data to all other servers. Each *node* helps its *neighbours* to have the updated copy of the database:

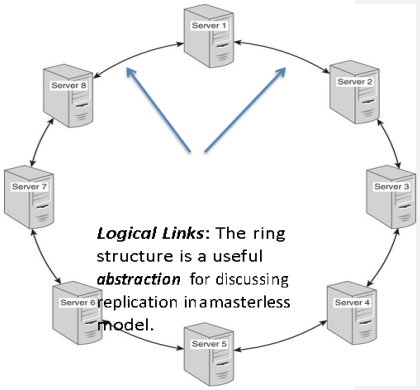


Figure 30- masterless replication.

1. A write request is sent to a random server of the cluster.
2. The server that responds to the write request is the master and it will send the update to his neighbours.
3. The neighbours will receive the write request, so they become master, and they do the same thing with their neighbours (*except the one from it received the update*).
4. Eventually, the update will propagate among the cluster.

Suitable for applications in which the number of writes may increase and achieve peaks! Example: the website which sells tickets to concerts.

Handling Replication: An Example

The database is divided in 8 chunks, when a server receives an update of a chunk, then he will forward the update to all the servers that contain the replicas of that chunk.
In this example every chunk is replicated three times; each time there is a write operation to one of the servers, the update is replicated to the three other servers holding its replica.

The rule for this example is: The chunk `K` *have its main copy in the server* `K`*, and it is replicated in the server* `K-1` *,* `K+1` *and* `K+2`*.*

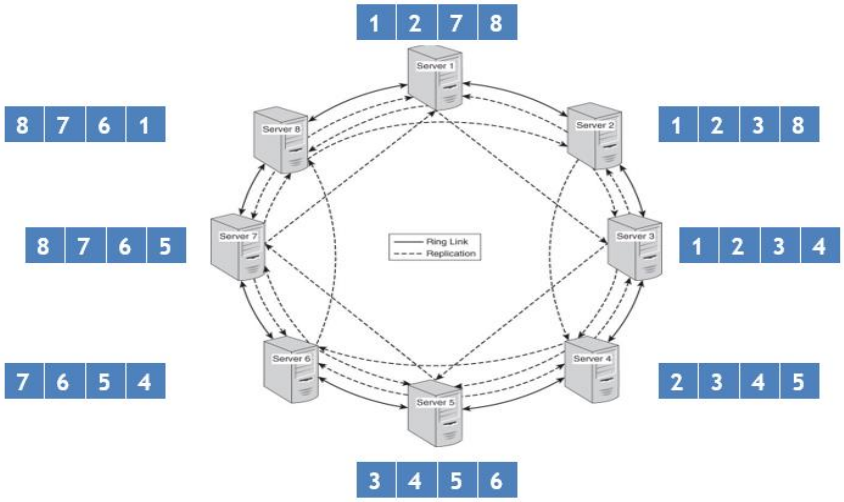


Figure 31- The example ring.

The ring structure is obviously a logic abstraction, in reality those servers are interconnected with a switch or a classic network. Usually, in a data centre, all the servers are connected to a single network hub and can communicate with each other.

How to construct a key: some guidelines

Keys in relational databases

In relational models, the *primary key* is used to retrieve a row in a table, and it must be *unique* because it identifies the tuple. When designing *relational databases*, using *meaningless* primary keys is a good practice (*like auto increment ids*).

The keys should be meaningful

Similarly, a *key*, in a key-value database is used to retrieve a value and must be *unique* in a specific bucket of a namespace. When designing *key-value databases,* adopting meaningful keys is *preferable*. A good practice is to build keys by embedding the entity information like: Name, identifier, or other attributes.

The keys should not be too long or too short

Long keys will use more memory, meanwhile short keys are more likely to lead to conflicts in key names.

The keys should follow a format

For each key, we may store in the database a specific value, so we have a key for identifying every attribute of the entity.

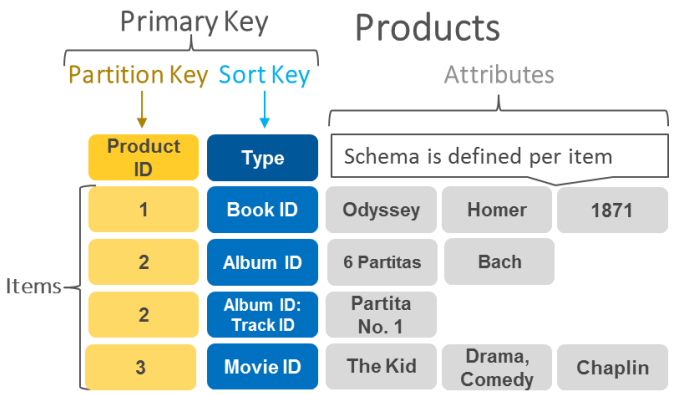


Figure 32 - A good format for the keys.

The number of attributes of an entity isn't always predictable, but we can add new attributes by simply changing the format of the key (the programmer decides it). Note that we don't have to define these keys in advance, it oversees the programmer to do this, we have a document with the list of the attributes and the key to an entity. It is normal to make some changes to the key format during the programming of the application. Before deciding the format of the key, you must consult the *documentation* for limitations on keys and values.

Single entity, no relationships.

A key should be defined on the entity itself not on the relationships with other entities.

The key should contain values that don't change.

Like the name or the date of birth, but not the amount of money in the bank.

Retrieve values on a key-value database

In general, keys may be any value/object (*since everything can be converted to a simple string*). But like in every memory, to access a certain value we need the address of the specific memory location in which this value is physically stored.

Hash functions

Hash functions are usually used to obtain the memory address (*or the server where it is stored*) starting from the key. Given a key in input, the function returns a hexadecimal number (*that seems to be a random value)*.
Note that (*depending on the hash function quality*) the values returned by hash functions (*given two different inputs*) may be not unique (*every Hash function has a certain degree of collision problems*).

Key	Hash Value
customer:1982737: firstName	e135e850b892348a4e516cfcb385eba3bfb6d209
customer:1982737: lastName	f584667c5938571996379f256b8c82d2f5e0f62f

Figure 33- Example of a hash function.

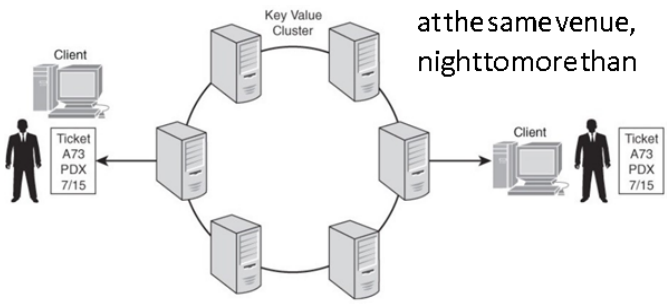
Hashing for Selecting Servers

Example 1

We would like to *balance* the write loads among the servers in a masterless replication architecture. How can I choose the server of the cluster where I will send the write request for that value?
For this case we can use a simple hash function, like the modulus. *For the sake of simplicity, let consider the ID of a specific server as the address to identify.* Note that the modulus of 8 has a range of values between 0 and 7.

$$ServerNumber = |Hash\ Value|_8$$

Example 2



Problem: We want to avoid selling two tickets for the same seat, in the same city.
In the picture, two fans want to purchase the same seat: *A73 at the Civic Center in Portland, Oregon, on July 15*. We can use the following key for the specific ticket:

Last Update: Thursday, 22 February 2024

```
TICK : A73 : CivCen : PDX : 0715
```

Note that the key is generated starting from the entity that I want to memorise; in fact, it contains specific values inside.
Anyone trying to purchase that same seat on the same day would generate the same key and the hash function will return the same value. So, there is no chance for another server to sell that seat because the server who receives the write request will see that there is another record for the same key.

Design characteristics for the key-value databases

When choosing a certain key-value database, consider the *trade-off of various features.*
A certain key-value database might offer *ACID transactions* but set *limits* on the dimension of keys and values.
Another key-value database might allow for large values but limit keys to numbers or strings.
Our *application requirements* should be considered when weighing the *advantages* and *disadvantages* of different database systems.

Operations allowed in Key-Value databases

In the following, we list the operations allowed in a Key-Value database:

- To *retrieve* a value by key.
- To *set* a value by key.
- To *delete* values by key.

These queries are always given by the DBMS itself, but if we want to do more, such as search for all addresses in which the city is “St. Louis,” we will have to do that with an application program.

Complex queries with key-value database

Complex queries must be made by hand by the programmers of the application. Here we can see a simple example of “complex query”.

```
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;
```

I define an empty list, then I make a loop where I retrieve all entities that I want.

Indexes

Some key-value databases incorporate *search functionality* directly into the database. A built-in search system would index the string values stored in the database and create an *index* for rapid retrieval. The search system keeps a list of words with the keys of each key-value pair in which that word appears.

Word	Keys
'IL'	'cust:2149:state', 'cust:4111:state'
'OR'	'cust:9134:state'
'MA'	'cust:7714:state', 'cust:3412:state'
'Boston'	'cust:1839:address'
'St. Louis'	'cust:9877:address', 'cust:1171:address'
	. . . .
'Portland'	'cust:9134:city'
'Chicago'	'cust:2149:city', 'cust:4111:city'

Namespaces

A namespace is a collection of `< Key , Value >` pairs. It is not allowed to have duplicate keys in the same bucket of a namespace; but we can have the same key in different buckets. of a namespace or in two different namespaces. In a semantic point of view, a namespace permits dealing with the same value but with multiple different keys. Namespaces are helpful when multiple applications use the same key-value database.

How to represent relationships?

In relational databases, the data are normalised, they expose the same characteristics for the data. In NO relational databases, the data are not normalised, they depend on the type of the application, if we have customers and orders, and there's a relationship between them, we can create two different applications that use the same data, but they are represented in different ways (*so we'll still work with only one namespace*)

Can I use two namespaces together?

With different namespaces we can't merge information from different namespaces, only data in the same bucket can be used together (*because they share the namespace part of the key*), but nobody will stop us from doing some simple string manipulation in the code and using the two namespaces.
If we want relationships management and join operation, we should exploit a relational database instead. Creating different namespaces and using them for join operations is the biggest mistake we can make in this course.

Data Partitioning

We divide data in partitions (*blocks, chunks, or shards*) and we spread partitions among the different servers that we have in our cluster.
We can have a namespace divided in some blocks, and we can have these blocks in different servers to distribute the load among servers
The main objective of partitioning data would be to evenly balance, both write and read loads, among servers.

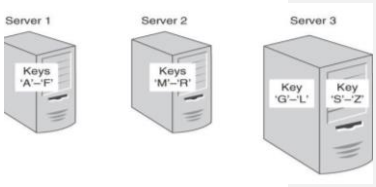


Figure 34 - Data partitioning.

Important properties of data partitioning

Different partitions may be handled by the different nodes. A node may contain more than one partition.

Add more node to the cluster

There are some strategies for data partitioning. If needed, *additional* nodes would be *easily* added to the cluster and data appropriately *relocated*.

Partition key

A partition key identifies the specific partition where the value has been stored.
Any key in a key-value database can be used as a partition key: for example, in the image above, the first letter of a key acts as the value of the partition key. Usually, hashing functions are adopted for identifying the specific cluster or partition.

Example of partitioning

In figure above we have 3 servers, and data is divided in 4 blocks:

- `A - F` → Server 1.
- `G - L` → Server 3.
- `M - R` → Server 2.
- `S - Z` → Server 3.

We need a partition key; in this key the partition key can be the key itself. The partition key is the value used to decide in which partition i can find the value, if the partition is decided from the key to the entity itself, then the key is the partition key.

The “schema-less” property

Unlike the relational model, we are not required to define all the keys and types of values we will use prior to adding them to the database.

Key-Value Database	
Keys	Values
cust:8983:firstName	‘Jane’
cust:8983:lastName	‘Anderson’
cust:8983:fullName	‘Jane Anderson’

We just need to update the application code to handle both ways of representing customer names or convert all instances of one form into the other. Consider that this type of database should be used for simple cases, where I don’t need controls or very strict constraints.

Clusters

A Cluster is a collection of independent servers.

Loosely Coupled Cluster

In Key-Value databases, clusters tend to be loosely coupled:

Minimal coordination with other servers in the cluster

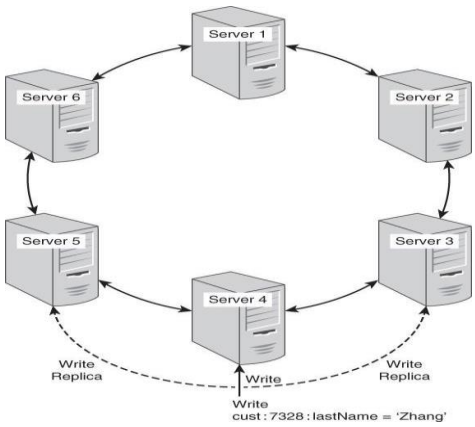
We usually ensure minimal communication in order to deal with some failure situations, or if we have a ring organisation, we can select a server with a query, but if the server is not available, another server will replace it.

The server is independent and completes many functions on their own

Each server is responsible for the operations on its own partitions and routinely sends messages to each other to indicate they are still functioning. When a node fails, the other nodes in the cluster can respond by taking over the work of that node.

Logical structure for organising partitions : Ring

Let consider a hashing function that generates a number from a key using the modulo.
Whenever a piece of data is written to a server, the replica of that data is written to the two servers linked to the first server (*neighbours*)



If Server 4 fails, then both Server 3 and Server 5 could respond to read/write requests for the data on Server 4. They can do it because they have the replica of data inside Server 4.
When Server 4 is back online, then Servers 3 and 5 can update it with the Write operation made while Server 4 was down.

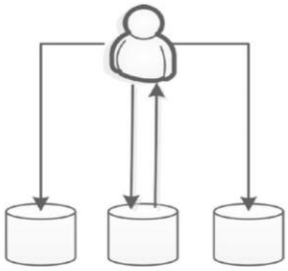
Replication

Availability is ensured by using replication. When I use replication, I have multiple copies of the data in the nodes of a cluster.

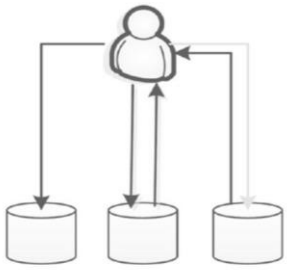
Trade-off for the number of replicas

A system is considered available if whenever we make a query, we receive an answer by the database.
The higher the number of replicas, the less likely we will lose data, but the lower will be the performance of the system (*in terms of response time*). The lower the number of replicas, the better the performance of the systems, the higher the probability of losing data.
During the replication process the system is not available.

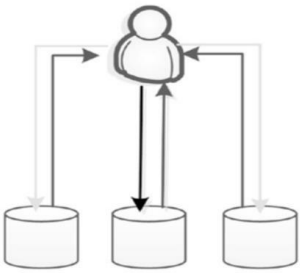
N = NUMBER OF REPLICAS.
W = NUMBER OF COPIES TO BE WRITTEN BEFORE THE WRITE CAN COMPLETE.
R = NUMBER OF COPIES TO BE READ FOR READING A DATA RECORD.



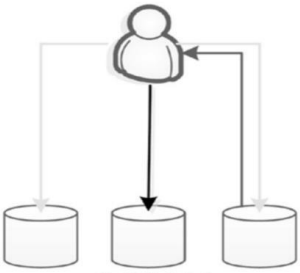
N=3 W=3 R=1
Slow writes, fast reads, consistent
There will be 3 copies of the data.
A write request only returns when all 3 have written to disk.
A read request only needs to read one version.



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



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



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

Choose the level of consistency

If we want replicas, we have to decide the level of consistency and the level of latency of our system. Depending on the requirements of our applications one of these 4 implementations above can be chosen.

Hash Mapping

In the example below, the Hash Value is a number in hexadecimal format.

Key	Hash Value
customer:1982737:firstName	e135e850b892348a4e516cfcb385eba3bfb6d209
customer:1982737:lastName	f584667c5938571996379f256b8c82d2f5e0f62f
customer:1982737:shippingAddress	d891f26dcdb3136ea76092b1a70bc324c424ae1e

Figure 35- Example of Hash function.

Hash Function Properties

One of the important characteristics of hash algorithms is that even small changes in the input can lead to large changes in the output. Hash functions are generally designed to distribute inputs evenly over the set of all possible outputs.
The output space should be large, this is useful when we are hashing keys.

Hash functions - example

Assume we have a cluster of 16 nodes and each node is responsible for one partition.
The key `'cust:8983:firstName'` has a hash value of
`4b2cf78c7ed41fe19625d5f4e5e3eab20b064c24`
would be assigned to partition 4.

The key `'cust:8983:lastName'` has a hash value of
`c0017bec2624f736b774efdc61c97f79446fc74f`
would be assigned to partition 12 (*c is the hexadecimal digit for the base-10 number 12*).

Even if we have small changes on the value in input of the function, we have a different value as output, there's no problem if we have similar keys distributed in servers, it depends on how we use that key.

Collision Resolution Strategy – Open Hashing

From a logical point of view, the table that projects a hashed key to the corresponding value may include a list of values. In each block of the list, also the original key must be present. Note that collisions are an uncommon event.

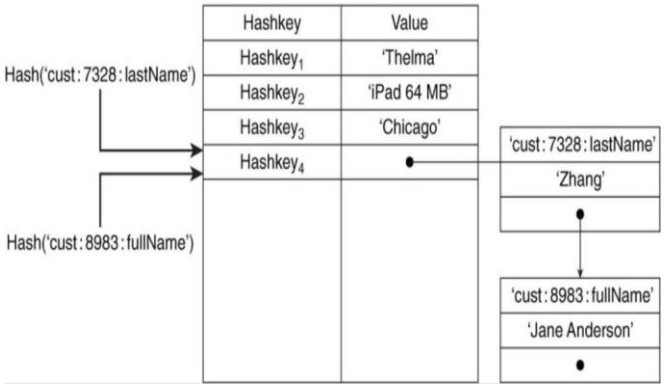


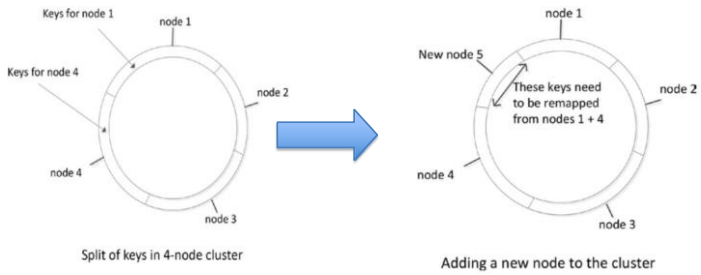
Figure 36 - The collision resolution strategy.

If we have calculated the hashing function of two keys, and we obtain the same value, we can use the open hashing, when there's a collision, we add the value in a list of values associated to a particular hash value.

Relocation of Data - Consistent Hashing

How do we deal with Horizontal scaling?

We need to change the hash function and relocate the data from the old partitions in the new ones. Consistent hashing is a strategy that ensures a good load balance among servers. When we remove a server, the data inside the partitions must be relocated, so we need to recompute the hash function; this is a problem because when you do this the system is not available. With consistent hashing we can remap any the keys of the new nodes and their neighbours.



Suppose that we have a ring of 4 nodes, if we want to add a new one, it will have its portion of key, we have to relocate the keys, and the key assigned at the new node is extracted from its neighbours.

Example of consistent hashing

Let's suppose we have an initial ring (*image below*) where we define all keys.
We suppose that the value of the hashing function goes from 0 to 2^x (*where x is the number of nodes*).

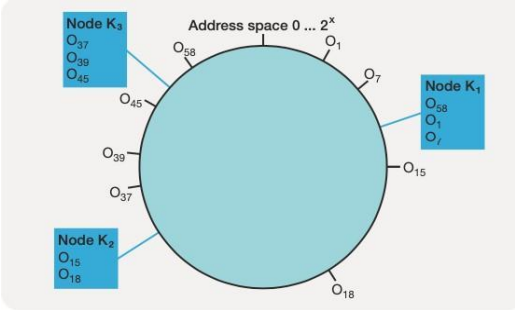


Figure 37- Example of consistent hashing, initial situation.

The node k_j is associated with a specific key (*object*) o_i is its **successor** *(in the case of K1, it is associated with O37, O39 and O45)* in the hashing/address space.
When we need to update the ring, we follow this rule: "*Moving the ring in reverse clockwise, the node takes all objects until it finds another node*".

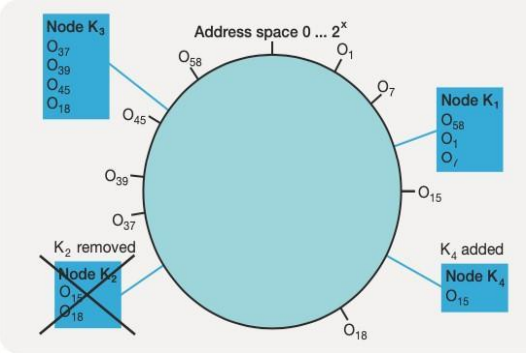


Figure 38- Example of consistent hashing, addition, and deletion of a node.

When we add a new node (*image above*), for example K4, the key 15 is removed from K2 and it is added to the node K4, when we remove node K2, the key 18 has been relocated in K3, it is a good solution to avoid relocated all the keys because to do that the system must be stopped.

Data compression for key-value databases

Key-value databases are *memory intensive.* Operating systems can exploit virtual memory management, but that entails (*comporta*) writing data from and to the disk.
One way to optimise the disk management is to use a data compression technique.

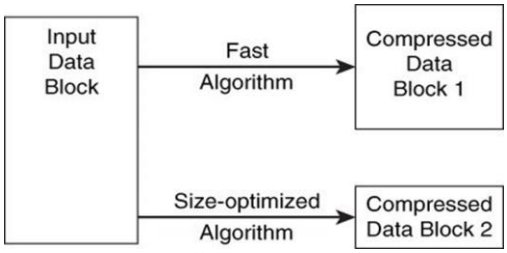


Figure 39 - The trade-off between two compression algorithms.

We are looking for a compression algorithm that ensures a trade-off between the speed of compression/decompression and the size of the final compressed data.

When choosing a Key-Value Database?

When should I use a key-value database?

- When I need very good performance.
- When we have a very simple data model and only "*one to many*" relationships.
- When we don't need very complicated analytics.
- When we don't have complicated relationships.
  - Because we can split and distribute the chunks of replicas in the clusters.
  - *Data splitting is not a good idea if I have to manage relationships and do join operations.*

Exercise: From relational to key-value

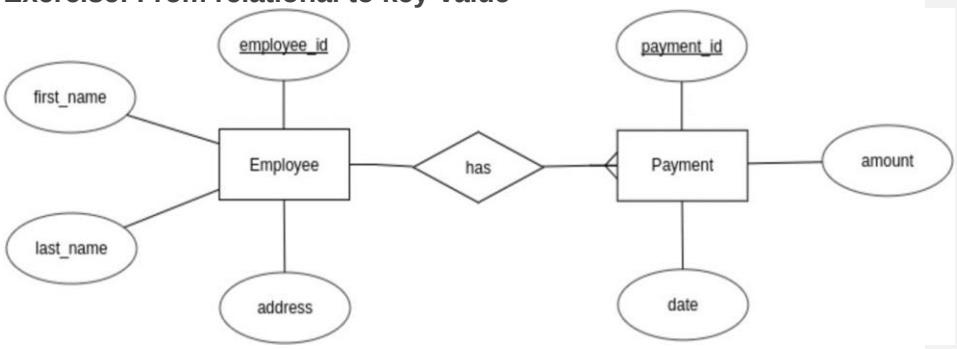


Figure 40 - The data model.

In this case we have an Employee associated with payment with a one-to-many relationship. In a relational database this one-to-many relationship would be handled by a foreign key, so with the functional dependency constraints.

Figure 41- The foreign key.

Translating the relational model to a key-value one

The first thing to do is to define the key format: take in mind that keys in a key-value database embed information regarding Entity Name, Entity Identifier and Entity Attributes.

What about NULL values?

If an entry doesn't have a value for an attribute (*for example the address*), it's an error to put this association with `null` value. The solution: Don't put an information if it isn't associated with a value.

Translate the Employee table

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"
```

Figure 42- The translation of the employee table.

For building the key we must concatenate the `$employee_id` with `$attribute_name`, and for every attribute we have the association with the value.

Translate the Payment table

We also have to manage the one-to-many relationship. In this case, we can define 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"
```

Figure 43 - Translation of the payment table.

The final result

We put the two entities in the same namespace and in the same bucket.
We put all in the same bucket, so, we don't need join operations, indeed the relationship between the two tables of the relational database is represented by unifying the keys in just one bucket.

```
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"

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"
```

Figure 44 - The final result.

KVDB Design – Tips and Tricks

Key pattern

A well-designed key pattern minimises the amount of code (*and the number of errors*) a developer needs to write to create functions that access and set values.
We compose a key in which the first part is the `id` of the entity, and the second is an attribute (*like the first name*).

```
cust:1234123:firstName: "Pietro"
custr:1234123:lastName: "Ducange"
```

Figure 45 - Wrong example of a key.

In the example above we are using two different prefixes for describing the same entity. This is wrong, you can just use `cust` to identify one user.

Get & Set methods

A class has attributes and methods, among the methods I expect to have the set and the get method for every attribute (*get for read and set for write*).
Using generalised get and set functions help improve the readability of code and reduces the repeated use of low-level operations, such as concatenating strings and looking up values.
Inside the key we can use range-based components to do queries, for example if we use a date as a string in the key, we can do a loop considering the interval of the date.

```
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
```

Figure 46 - Example of get and set methods for a key-value database.

The get method, we create the key which is created by concatenating the `id` of the customer and the wanted attribute name. The get function returns the value associated with that key inside the bucket called `AppNameSpace`.
The set method creates the key with the get function and then assigns a value at a specific attribute.

Dealing with range of values

Query: retrieve all customers who made a purchase on a particular date. First, we should add some info into the key:

`cust : p_date : p_id : custId`

Embedded in the key there is:

- The date of purchase (`p_date`), to quickly obtain information about the customer that made that specific purchase.
- The product id (`p_id`).
- The customer id (`custId`).

Then I loop the array looking for the keys that contain the value of the 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);
```

Figure 47- The code for loop the array of values.

Too much code? Wrong database model!

1. We initially create an empty list of customers.
2. We initialize the range (`v_rangeCnt`) to `1`, it is the first element of the list (*it's the counter of purchases*).
    a. `1` *it means: "that customer was the first to make a purchase in that date".*
3. We create the key by concatenating `cust`, the date, the incremental id, and the customer id.
4. We make a loop in which:
    a. Check if the key exists with `exists(v_key)`
    b. If the key exists, we retrieve the values `.append(myAppNS[v_key])`
    c. We increment `v_rangeCnt`.
    d. We update the key.
5. Finally, we return the customers list.

Sometimes it is requested in the exam to write a code like this.

Retrieve whole entities and complex structures

We want to retrieve the whole customer. So, we will use the get method for retrieving all attributes I need and then return them as a list of values or a JSON string.

```
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);
```

Figure 48 - Retrieve the full name and the address of a customer.

Above we can see the method in which we retrieve information about the full name and the address of a customer.

1. We pass the id of the customer as parameter.
2. We get the values of attributes by using the get function.
3. We create the wanted composed attributes
4. Then we return a list of full names and addresses.

How many accesses to the database?

In this case we make 6 accesses to the database (*because we called the* `getCustAttr` *6 times*), note that a key-value database usually stores the entire list together in data blocks, thus just one block in the disk will be accessed rather than six.
But when we hot horizontal scaling: those six keys could be on different servers, so with the access time I will have latency too.

How can I improve the situation? Unique key

When I am designing the database, I must think about HOW the data will be used, because the database should be organised for the query well 'going to do.
If I realise that I always use this function (*that invokes the get method for six times so I will do six access to the database*) then I should define a key that contains all this info together. So, I can retrieve the whole entity with only one look-up, but this redundancy has a cost:

- I must maintain the redundant information always consistent.
- The higher is the dimension of the object, the higher is the probability that this object is distributed in different servers and in different blocks of the disc.

Put a JSON file inside the key-value database

The higher is the dimension of the object, the higher is the probability that this object is distributed in different servers and in different blocks of the disc. In the example below we can see one of those cases.

```
{
  'custFname': 'Liona',
  'custLname':  'Williams',
  'custAddr' :  '987 Highland Rd',
  'custCity' :  'Springfield',
  'custState': 'NJ',
  'custZip'  :  21111,
  'ordItems' [
        {
            'itemID' : '85838A',
             'itemQty' : 2 ,
            'descr' : 'Intel Core i7-4790K Processor
              (8M Cache,
   4.40 GHz)',
        'price:' : $325.00
        } ,
        {
         'itemID' : '38371R',
         'itemQty' : 1 ,
        'descr' : 'Intel BOXDP67BGB3 Socket 1155, Intel
          P67',
            CrossFireX & SLI SATA3&USB3.0, A&GbE, ATX
            Motherboard',
        'price' : $140.00
         } ,
         {
        'itemID' : '10484K',
        'itemQty' : 1,
        'descr' : 'EVGA GeForce GT 740 Superclocked Singl
          Slot 4GB
            DDR3 Graphics Card'
             'price': '$201.00'
         } ,
```

Figure 49- A very complex entity, where we put a JSON object inside an attribute.

We are not sure that information is stored in the same block of the disc, and even if it is stored in the same server, so with key-value DB we don't have to use complicated data structures.

Limitation and cons of a KVDB

The only way to look up for values is by the key, so it's not ideal when I must do many range queries.

We may have some DBMS for key-value that offers some functionalities, usually as basic methods offered, we can search by key, delete by key and update by key.

The problem is that there are not standard languages like in SQL, to do those operations.

Case Study – Mobile App configuration

We consider a mobile application that track customers shipping.

About each customer we store:

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

Most of the operation done in the database are reads.

Customer Information:

```
TrackerNS['cust:4719364'] = {'name':'Prime Machine, Inc.',
  'currency':'USD'}
```

Dashboard Configuration options:

```
TrackerNS['dash:4719364'] =
  {'shpComp','shpState','shpDate','shpDelivDate'}
```

Case Study – Restaurant

We want to exploit the system for reservations.
Let's suppose that in one date `"yyyy-mm-dd"` we can have only one reservation for table.
For each table we have the maximum number of people that can seat in that table, it is a fixed value.
If you make a click on a table booked you have some information about the name and the telephone number of the customer, the DB maintains this information only for the time needed for the reservation and not in a persistent way, on the screen we have also the date.
For the write operations, we also have tables, and we can select a table and insert the seats, the name of the customers and the telephone number, there's also the date.

How can we format the key?

`Res : $id_table : $date : attribute`

We can't have two reservations in the same table at the same date, so those two values can identify the reservation (without exploit a reservation_id).

Which attribute can I put?

We need to keep the reservation description, so we can add a new attribute called `$res_des`.
`$res_des` is like `{Customer name , Cell number , Number of people}`

At the end we got those entities

```
Res : $id_table : $date : $res_des
Tab : $id_table : $maximumPeople
```

Mind the requirements! the system is used ONLY for make and see the reservations , for check a reservation I have too many details.

Can I organize the database like a calendar?

So, I add another key, that show a different information:
`Res : $id_table : $date : $status`
If `status = 0` then the table for that date is available; otherwise, the attribute tells me the number of people of the reservation.

Document Databases

Document databases are non-relational databases that store data as structured documents, usually in XML or JSON formats. They ensure a high flexibility level and allow also to handle complex data structures (*despite KVDB*).
Document databases are schema-less, but they allow complex operations, such as queries and filtering. Some document databases also allow ACID transactions, like MongoDB.

What do we need?

Complex analysis

Key-value databases ensure fast response and flexibility, but we would need to make complex analyses maybe using some other attribute than the key.

High Flexibility

We have to ensure high flexibility but, in this case, we want to use complex data structures too, we can define the attributes each time we need, whenever we need because the architecture is schema-less.

XML Documents

XML stands for eXtensible Markup Language. A markup language (*like HTML*) specifies the structure and content of a document but also the meaning (*a document is a file which contains some text*). Tags are added to the document to provide the extra information, for example it can contain an attribute of an object.

Properties of XML

- XML tags give a reader some idea what some of the data means.
- XML can represent almost any form of information.
- XML was used a lot in the past for encapsulation and transmission of data, between client and server.
- XML is mostly used in web2.0 because together with CSS gives the possibility to implement new generation websites, XML was also used a lot in the past decades for encapsulating data that are transmit from a client to a server and vice versa, for example with data interchange protocols like SOAP.
- XML is used for formatting documents such as word documents, excel documents and PowerPoint documents.

Pros of XML

XML is text (*Unicode*) based, so it is easy to transmit, and then we can copy text from a file to another, so it has high modularity, we can create separate modulus and merge them together.
One XML document can be displayed differently in different media and software platforms.
XML documents can be modularized, so its parts can be reused for other XML documents.

```
<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>
```

Figure 50 - Example of XML document.

The code in the image describes three albums. Each album contains: the title, the name of the artist and all the names of all tracks inside the album. Inside a track we can put some properties, for example the length of the song. May exist several APIs that allow us to read this file, the docx file in Microsoft Word is an XML file too.

XML Documents: Use cases

- XML (*like CSS*) allows separating data and format of the document.
- XML is the basis for many data interchange protocols and was a foundation for web service specifications such as SOAP (*Simple Object Access Protocol*).
    - The SOAP protocol is the ancestor of the Restful API.
- XML is a standard format for many document types, including word processing documents and spreadsheets (*docx, xlsx and pptx formats are based on XML*).

XML Ecosystem

XML includes or actually included (*we aren't sure that it includes all the features in the list*).

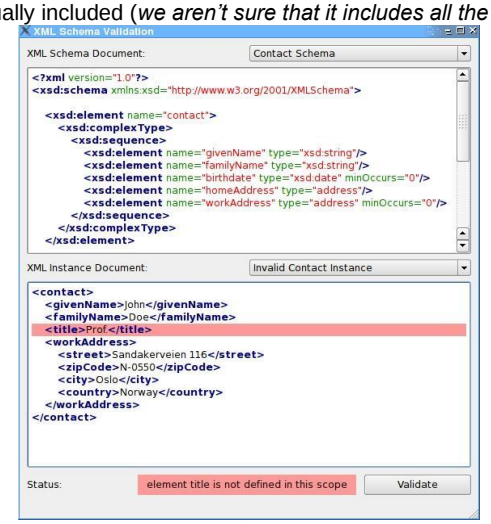


Figure 51 - IDE for XML documents.

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.

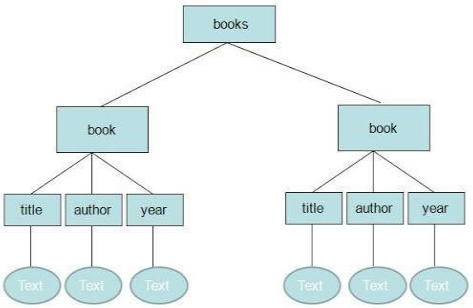
XML schema Document

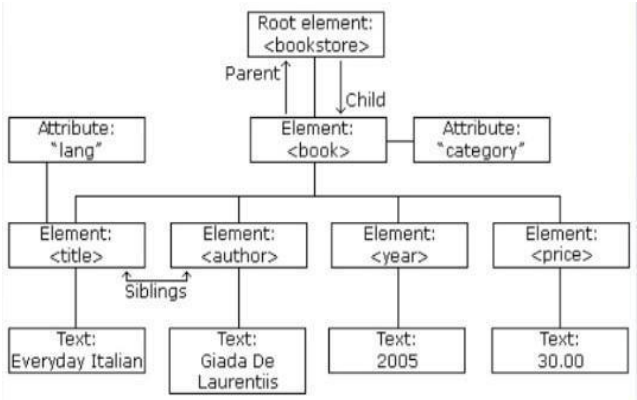
A special type of XML document that describes the elements that may be present in a specified class of XML documents. Often used for validating the structure of an XML file.
In the picture on the right, I can see an example of usage of an XML scheme. Each information has to be embedded in an open and close tag; therefore, each tag could have other properties. Every XML document represents an object.
In this example, if we have a telephone book, each element is a contact and it must have some attributes, if you create an attribute prof, you'll have an error because prof is not included in the list of the attributes defined before.

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.
An XML file can be stored as a binary file, a file is a set of bytes, we can represent an XML like a tree in memory and in the disc.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="web" cover="paperback">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```





Do we have XML available databases?

Some frameworks proposed in the past and a general architecture of XML based databases had this three levels architecture:

XML Databases

XML databases: platforms that implement the various XML standards such as XQuery and XSLT,
They provide services for the storage, indexing, security, and concurrent access of XML files.
The storage layer is for storing persistently data, in this layer we have indexes too.
The storage layer can be accessed by a set of applications engines which provide services to the clients.
XML databases did not represent an alternative for RDBMSs.
On the other hand, some RDBMSs introduced XML, allowing the storage

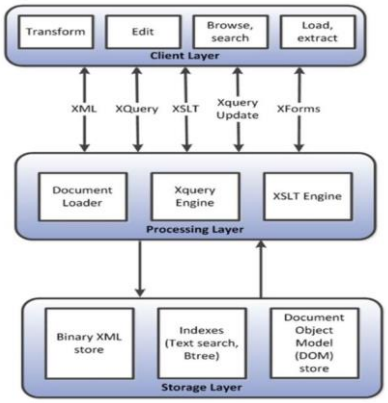


Figure 52 - Structure of a XML database.

Main Drawbacks of XML

- XML tags are verbose and repetitious, thus the amount of storage required increases, due to, for example, the close tag.
- XML documents are wasteful of space and are also computationally expensive to parse.
- When I have too much stuff in an XML file, it becomes very big, so in case of a network transmission it can be a problem.

Is XML still used?

XML format and language was very useful some time ago with the web2.0, however XML is too verbose and with a lot of repetitions: every tag occupies memory, so it is computational expensive to do operations with objects in XML files.

Are XML databases still used?

In these years, XML databases are used as content-management systems: collections of text files (*such as academic papers and business documents*) are organised and maintained in XML format (*like word files*).

JSON (*Javascript Object Notation*) documents

Thanks to its integration with JavaScript, a JSON document has been often preferred to an XML document for data interchanging on the Internet. An example of JSON document:

```
{
  "nome": "Mario",
  "cognome": "Rossi",
  "eta": 30,
  "indirizzo": {
    "via": "Via Roma, 123",
    "citta": "Milano",
    "cap": "20100"
  },
  "telefoni": [
    {
      "tipo": "cellulare",
      "numero": "123-456-7890"
    },
    {
      "tipo": "casa",
      "numero": "987-654-3210"
    },
    {
      "tipo": "ufficio",
      "numero": "555-123-4567"
    }
  ]
}
```

A JSON file is a collection of key-value associations, the key is often a name, so there are name-values associations. A JSON database is similar to a KVDB: indeed “web” and “database” in the image on the right can be seen as two keys and the values of each key are vectors.

Comparison of JSON and XML

Similarities

- Both are human readable.
- Both have very simple syntax.
- Both are hierarchical.
- Both are language independent.
- Both are supported in APIs of many programming languages.

Differences

- Syntax.
- JSON is less verbose.
- JSON includes arrays.
- Names in JSON must not be JavaScript reserved words.

Features of JSON Databases

Data is stored in JSON documents and High flexibility

A document is the basic unit of storage (*in relational DB the basic unit is a table*).It includes one or more key-value pairs and may also contain nested documents and arrays (*for a more complex hierarchical structure*). We can have denormalized data.

Collections and Polymorphic Scheme

A collection is a set of documents sharing some common purpose (*given by the developer*). The documents in a single collection may be different, but they are used for the same purpose.

Schema less

You don't have to have predefined document elements. Whenever we want a schemeless database, the designer of the application can write functions for removing, inserting, and updating values without modifying the database. We may decide to add a new document without modifying the others.

Document databases: data modelling

Schemeless property

A schema is a specification that describes the structure of an object, such as a table. Every object of that group must respect the predefined scheme.
Data modellers must define tables in a relational database before developers can execute code to add, remove, or update rows in the table.

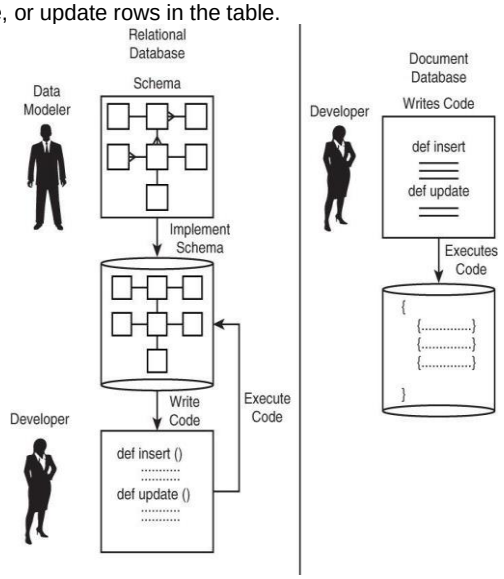


Figure 53 - The schemeless property advantages.

Document databases do not require this formal definition step. Developers can create documents in collections by simply inserting them.

Pros of a schemeless database

High flexibility in handling the structure of the objects to store.

Cons of a schemeless database

Nobody will check if I respect constraints given by the requirements. In fact, a relational database automatically does checks on the type of values and on relationships, because we define the schema before inserting the data.

Some considerations on the third normal form

A document database could theoretically implement a third normal form schema. Tables may be "*simulated*" considering collections with JSON documents with an identical pre-defined structure.
So, with a document database, you can simulate a third normal form schema, but in the project this trick cannot be exploited (*or the project will be rejected*).

How to deal with relationships among entities on a JSON Database

Document Embedding

Document databases usually adopt a reduced number of collections for modelling data. The red part is the embedded document.

```
{
  "name": "Mario",
  "surname": "Rossi",
  "dateOfBirth": "2000-08-08",
  "address": {
    "road": "Via Roma, 123",
    "city": "Milano",
    "cap": "20100"
  },
  "phoneNumbers": [
    {
      "type": "Mobile",
      "number": "123-456-7890"
    },
    {
      "type": "Home",
      "number": "987-654-3210"
    },
    {
      "type": "Office",
      "number": "555-123-4567"
    }
  ]
}
```

Nested documents are used for representing "*one to many*" relationships among different entities. Like in the code above, a person can have multiple phone numbers, and each phone number is represented with a document embedded in the main person document.

Document Linking

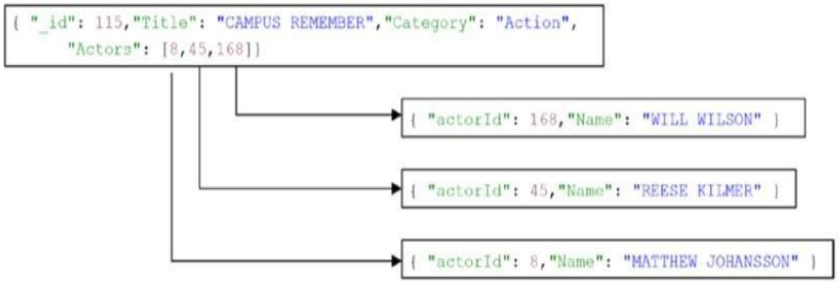


Figure 54 - Example of document linking.

An array of actor IDs has been embedded into the main film document.
This kind of strategy can be used in some particular situations, especially when we got "*many to many*" relations. The retrieved IDs can be used to retrieve the documents of the actors (*in another collection*) who appear in a film.

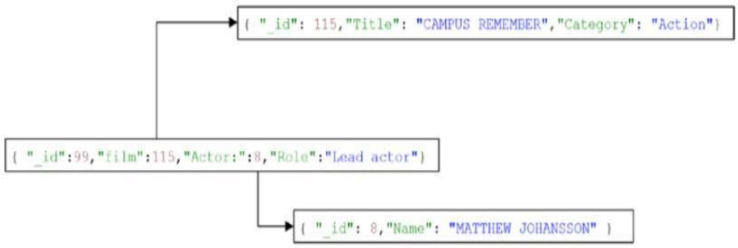


Figure 55- Example of document linking.

We are rolling back to the third normal form!

In fact, 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 (*embedded document may be updated*).
When modelling data for document databases, there is no equivalent third normal form that defines a "*correct*" model.
In a schemeless database the queries drive the approach to model data, if I have to deal with many and complex relationships, I should choose a relational one instead.

How to model a "one-to-many" relationship

```
Person{
  "name": "Mario",
  "surname": "Rossi",
  "dateOfBirth": "2000-08-08",
  "address": {
      "road": "Via Roma, 123",
      "city": "Milano",
      "cap": "20100"
  },
  "phoneNumbers": [
    {
      "type": "Mobile",
      "number": "123-456-7890"
    },
    {
      "type": "Home",
      "number": "987-654-3210"
    },
    {
      "type": "Office",
      "number": "555-123-4567"
    }
  ]
}
```

The basic pattern is that the *one* entity in a "*one-to-many*" relation is the *primary document (in this case the* `person`*)*, and the *many* entities (in this case the `phoneNumbers`) are represented as an array of *embedded documents*. This is not fully normalised (*often the embedded documents are part of data in another collection and do not depend on the key to the main entity*).

How to model a "many-to-many" relationship

In this example a `student` can be enrolled in many courses, and a `course` can have many `students` enrolled in it.

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

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

We must take care when updating data in this kind of relationship. Indeed, the document DBMS will not control the referential integrity.
In this case is preferable to use Document Linking because when a professor makes the access, he can retrieve all the document related to him; so, using document linking improves the performance of the application. Document embedding it's not preferable because I could put too much stuff in only one object.

How to model hierarchies

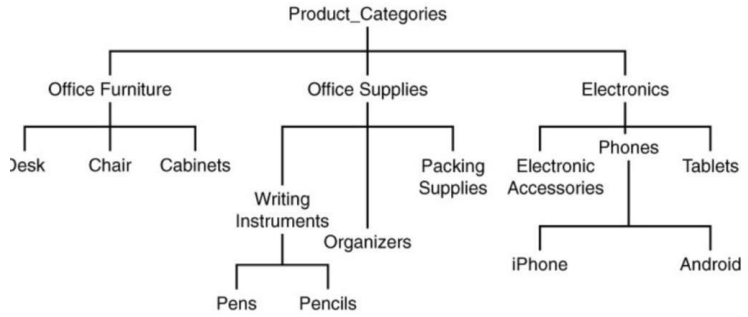


Figure 56 – Example of a hierarchy.

We can exploit three types of solutions:

Parent-reference solution

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

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

Child-reference solution

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

```
{
   {productCategoryID: 'P01', name:'Product Categories',
```

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

List of Ancestor solutions

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

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

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

Exercise - Trucks Company

Trucks in a company fleet must periodically transmit this information:

- Current location.
- Average fuel consumption
- Other metrics

The transmission is done every 3 minutes to a fleet management database (*one-to-many relationship between a truck and the transmitted details*). Trucks are moving only between 8:00 AM and 6:00 PM.

Before modelling, let's do some computations

We consider 20 transmissions per hour, and the truck works for 10 hours every day so at the end of the working day database will include 200 new documents for each truck. We can create a document for every transmission: each document contains:

```
Update{
  "truckID": 23,
  "driverID": "MarioRossi00",
  "timestamp": "2023-08-08T17:15:00",
  "AVGFuelConsumption": "4 mpg"
  "Location": <coordinates>
 }
```

Every three minutes the client creates this document and transmits it to the company server. This information has a lot of redundancies: If we look two consecutive update the red fields are always the same.

Alternative solution

Put the variable information is embedded into `operational_data`, in this way it is easier to retrieve information in this situation rather than the previous case in which we had to scan all the information.
This solution may produce enormous documents! mind that the maximum size of the single document has a limit (*in MongoDB the limit is 16Mb per document*).

```
Update{
  "truckID": 23,
  "driverID": "MarioRossi00",
  "date": "2023-08-08"
  Operational_data{
        "time" : "17:15:00",
        "AVGFuelConsumption": "4 mpg"
        "Location": <coordinates>
      }
 }
```

Planning for mutable documents

Documents relocation

It's important to have an idea of the maximum dimension of the documents in my database.
When a document is created, the DBMS allocates a certain amount of space (*equal to initial space of document + fixed more memory*) for it into the disk.
If the document grows more than the allocated space:

1. The DBMS search (*and possibly free*) another space in the disk.
2. Relocate (*cut and paste*) the document.

This operation is transparent at high level, but it is very time-consuming.

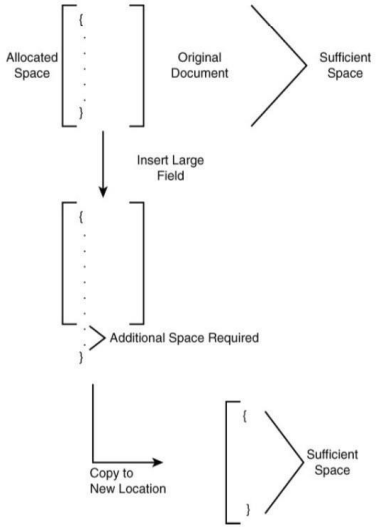


Figure 57 - Reallocation of a document.

A solution for avoiding moving oversized documents is to allocate sufficient space now in which the document is created.

Life cycle of a document

We must plan, if possible, the strategies for handling its growth. Obviously, it strongly depends on requirements. For example (*every year, delete the reservation of the previous year*).

Indexing a document database

To avoid the entire scan of the overall database when I search a certain document, DBMSs for document databases (*for example MongoDB*) allow the definition of *indexes*.
Indexes are special data structures that store a value of one (*or more*) field, ordered by the value of the field (*or fields*) in order to permit a direct access to it (*faster read*).
Often an index is implemented like a set of pointers to documents.

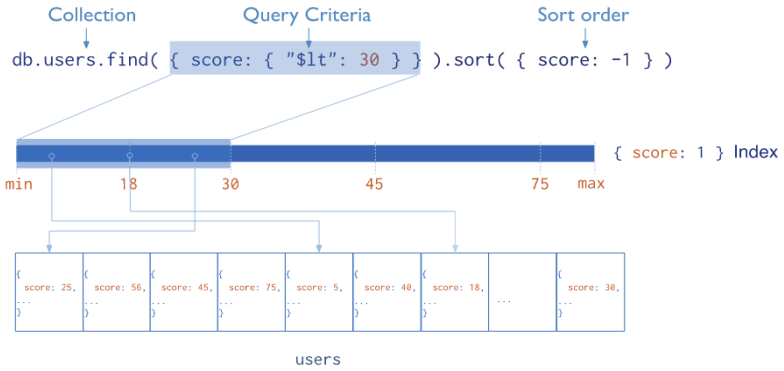


Figure 58 - Example of an index.

When is an index exploited?

An index is exploited depending on the match criteria. If the index is created on the field A and the match criteria of the query regard the field A, then the index will be exploited.

Applications and indexes

There are two types of applications.

Read-Heavy Applications

The use of several indexes allows the user to quickly access the database. For example, indexes can be defined for easily retrieving documents describing objects related to a specific geographic region or to a specific type.

More indexes → Faster read operations → Slower write operations

Write-Heavy Applications

The truck information transmission (*each three minutes I must update something inside the database*) is a typical write-heavy application.
The higher the number of indexes adopted the higher the amount of time required for perform a write operation, because when I modify the data all the indexes defined on that data must be updated (*and created at the beginning*).

Less indexes → Faster write operations → Slower read operations

Trade-off solution

The number, and the type of indexes to adopt must be identified as a trade-off that depends on the estimated number of reads and writes.

Transactions processing systems

As we saw in the indexes part, in general we can't optimize reads and write at the same time.

What do we do?

These systems are designed for fast write operation and targeted reads, as shown in the figure below:

- ***OLTP:*** OnLine Transaction Processing.
- ***OLAP:*** OnLine transaction Analytical Processing.

I can have two databases: one tuned for improving reads and the other for improving writes operations. Usually, the read database it's a document database and the write one is relational.

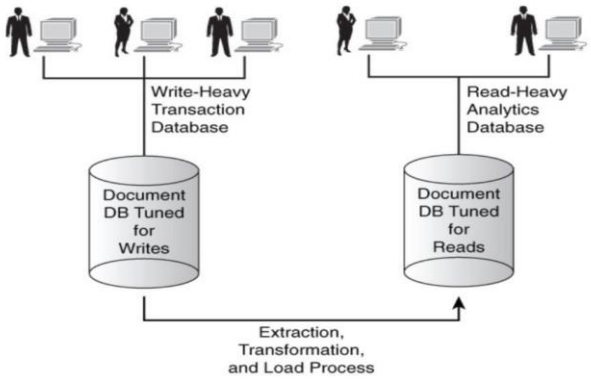


Figure 59- A transaction processing system.

How do these two databases cover data among them?

At night (*low level of load*) I can update the read-tuned one with the new information contained in the write-tuned one.
However sometimes data doesn't need to be processed in real time (*maybe we don't use analytics every hour*), for example at a certain interval we may extract and transform information from Analytic DB and put them into Document DB in order to do faster analytics.
We may have different granularity of an event entity; we may not need to store in the Document DB each data that we have on the Analytic DB. We can enforce CA on the server on the left and only AP on the right server (CAP theorem).

Partitioning

Vertical Partitioning

It's a technique used for improving (*relational*) database performance by separating columns of a relational table into multiple separate tables. So, from a table I obtain two different tables.

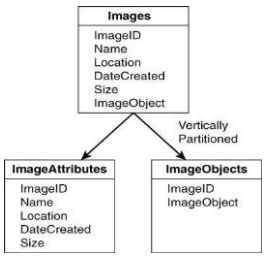


Figure 60- Partitioning in a relational database.

Horizontal Partitioning - Sharding

Sharding or horizontal partitioning is the process of dividing data into blocks or chunks. Each block, labelled as "*shard*", is deployed on a specific node (*server*) of a cluster (*it is like the partitioning of the disc*). Normally each node can contain only one shard, but in case of data replication, a shard can be hosted by more than one node.

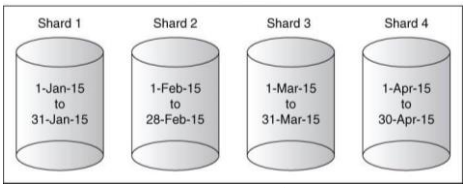


Figure 61- Example of sharding.

Pros of sharding

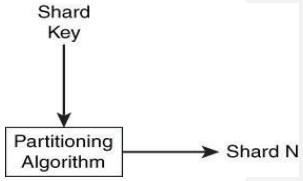
- Handling heavy loads and the increase of system users.
- Data may be easily distributed on a variable number of servers that may be added or removed by request.
- Cheaper than vertical scaling.
- Combined with replications, ensures a high availability of the system and fast responses.

How can we create shards?

We have to create a shard key which contains some attributes, only an atomic field can be chosen as a shard key. We may use the name, the date etc.

Shard Keys

We must select a shard key and a partitioning method.
A shard key is one or more fields that exist in all documents in a collection (*the value of the attribute doesn't have to be unique*), that is used to separate documents.
Examples of shared keys may be like (unique document ID, Name of a person, Date of birth, creation date, Category or type, Geographical region,).



Partition Algorithms

There are three main categories of partition algorithms, based on:

- ***Range***
  - If all documents in a collection had a creation date field, it could be used to partition documents into monthly shards.
- **Hashing**
  - Can be used to determine where to place a document.
  - Consistent hashing may be also used.
- **List**
  - 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.

Collections

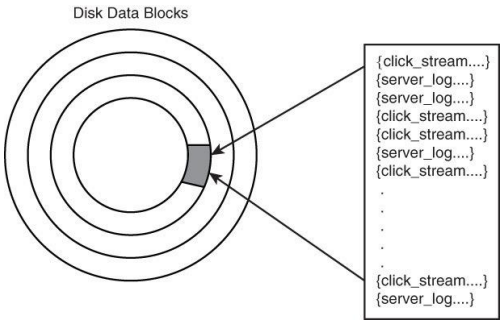
A collection is a set of documents sharing a collective purpose.

From MongoDB Manual: "*MongoDB stores documents in collections. Collections are analogous to tables in relational databases*."

What is happening at a low level?

The DBMS will put documents belonging to the same collection inside the same disk block, so retrieving all documents of a single collection is a pretty fast operation.

In the figure below all the information is put together in the same disk block, this is nice if we use that information together (*for instance doing some analytic*). But if you use this information separately then it's inefficient because each time you read a block (*reads in disk are made only on blocks and not on single records*) you must use a filter for retrieve only the data you want.



What if I use an index?

If we want faster read applications, we can use indexes. Indexes can solve the problem of having heterogeneous information in the same block, and this avoids making a filter after retrieving the block. If we have a large collection, it is good to have a lot of indexes.
But the problem remains, so indexes can help but separating information would be better because if there is a lot of data and they are very different, indexes then become very inefficient.

Manipulate a collection with code

In general, a code written for this purpose should have:

- A substantial amount of code that applies to all documents.
- Some amount of code that accommodates specialised fields in some documents.

High-Level Branching

Let's suppose that in the collection "log" we got two types of documents (click_stream and server_log).
We may have two applications, one for the `server_log` and one for `click_stream`, so the types of events are separated.
In this case we have the desire to separate stuff (*even inside the code of the application*), so we should create separate collections. In this case we have a high-level difference between the two objects.

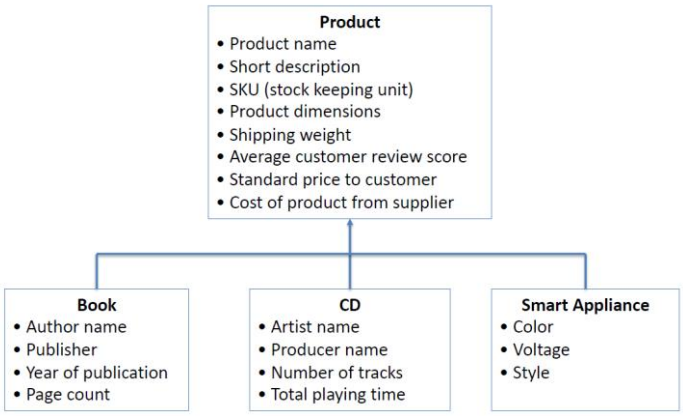
```
High-Level Branching
   doc.
If (doc_type = 'click_stream'):
    process_click_stream (doc)
Else
    process_server_log (doc)
```

Low-Level Branching

We have a document `book` described by the parameters.
If `doc.ebook` is `true`, then only `book.descr` is different.
In this case we don't have to create another collection because there aren't any high-level differences between those two objects.

```
Lower-Level Branching
book.title = doc.title
book.author = doc.author
book.year = doc.publication_year
book.publisher = doc.publisher
book.descr = book.title + book.author + book.year + book.publisher
if (doc.ebook = true);
   book.descr = book.descr + doc.ebook_size
```

Queries on a Document Database



The product is specialised into a book, a CD, or a Smart Appliance.

How to define entities

How can we model this UML class diagram? We must know how many collections are required in our system, and how many fields must be included in each entity, both mandatory fields and optional fields.
How is the third-normal form? We will have four tables and we refer to the specific specialisation with the ID.
How many collections could you consider? It depends on the application.

When we define entities, we must follow the definition of the queries.

Let's suppose that we have these queries of functional requirements:

- *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?*

Specialisation and collections

We never read in the requirements about the specialisations of products, in the requirements we have only products in general.
In this case we must maintain all the products in the same collection.

Then we define one additional field in the product (*which is a document embedded on the first one*) that contains all the information about its specialisation.
Usually, we have mixed types of functional requirements.
The difference is that now we don't need to put our hands into a database, we can use structures defined inside the code.

If we separate entities in different collections, and the number of entity types grows then the number of collections would become unwieldy.

Normalisation vs Denormalization

Which is better? It depends on the application that I am creating.
With normalised data I limit anomalies, but I could need join operation, so I could add latency to operations and complexity to the code.
Without normalised data I could have some redundancies (*and probably, some anomalies too*) but the latency decreases because I don't need to do join operations and the code will be simpler.
But redundancies must be updated.

Columnar Databases

They are strictly connected to relational databases. A column-oriented (*or simply columnar*) is a DBMS that stores data tables by column rather than by row (*remember that a column contains values of a certain type*).

Pros of columnar databases

- More efficient access to data when we are only querying a subset of columns (*by eliminating the need to read columns that are not relevant)*.
    - Typically an analytic is computed using only one attribute, so retrieving all the values of a certain column requires one or few more access to the disk.
- More options for data compression.

Cons **of columnar databases**

Typically less efficient for inserting new data: indeed if you have to insert a new row, you have to add 1 value in each attribute column, so you may have to do multiple accesses to the disk.

Similarities between Columnar and Row Databases

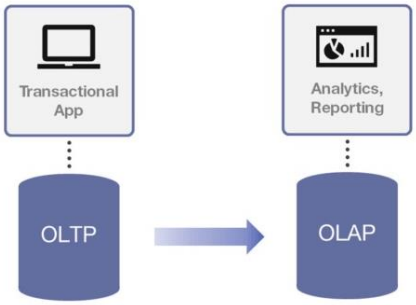
- Both can use traditional database query languages like SQL to load data and perform queries.
- Both can become the backbone in a system to serve data for common extract, transform, load (*ETL*) and tools.

OLTP vs OLAP

OLTP

Regards the representation of the relationships using the relational model.

- Oriented to handling ACID properties of transactions.
- Normalised data, so many tables.
- High volume of transactions.
- Simple and fast queries.
- Fast processing of write operation (*few indexes or none*).
- Queries like: "*Give me the list of the users who bought X*"



OLAP

- Oriented to fast analysis of data.
- Denormalized data, so fewer tables.
- High volume of data.
- Complex and slow queries.
- Fast processing of read operation (*many indexes*).
- Queries like: "Tell me *how many* users bought *X?*"

Row data organisation

A file is a sequence of records (*list of bytes*) organised in a certain way (*the way to read the file is determined by the format of the file*) in the disk.
For relational databases the rows of the tables are stored on the disk, this is nice for OLTP.

CRUD and Queries

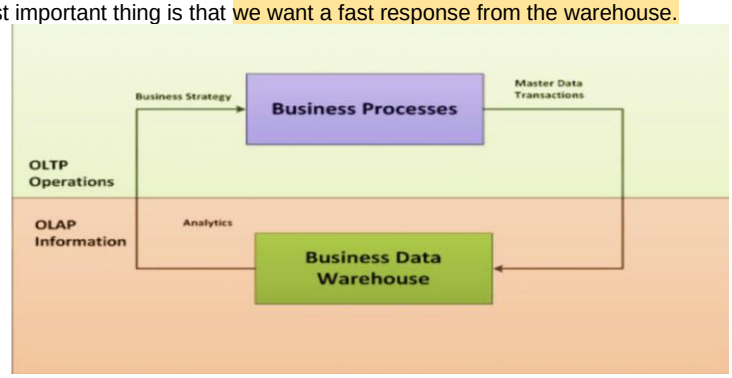
CRUD **C**reate, **R**ead, **U**pdate and **D**elete operations.
In the "*record-based*" era, CRUD operations were problematic in terms of time response, but the latter was not a critical issue in that time because systems were mostly OLTP.
When the OLAP system assumed a big relevance, thus the time response became very important, so the record-based organisation started to give some problems.

Data Warehouse

A data warehouse is a data management system designed to support business intelligence activity (*big and complex analysis operations).* In the figure below there's the typical business process, for example:

1. When we are selling something, a transaction occurs.
2. That transaction generates useful data.
    - We store them in the warehouse in some way.
3. Using those data, we can generate analytics for the management.

The most important thing is that we want a fast response from the warehouse.

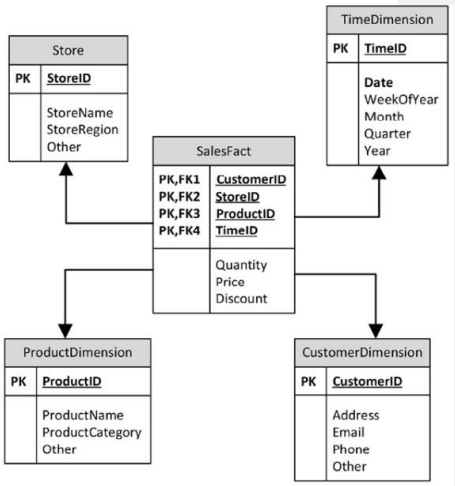


A typical data warehouse often includes:

- Relational Database for store and manage data.
- Solutions for optimising the response of analytics.
- Analysis, data mining functionalities.

Star Scheme

It is a relational solution where central large "*fact tables*" are associated with numerous smaller tables. A star scheme is a fully normalised model of data (*redundancies accepted and sometimes not all attributes depend on the Primary Key*), so this is not the classical third normal form. It is a denormalized data representation that optimises the response time of some queries. All other tables have a dependency on the central one, so the entire scheme looks like a star (*from this the name of the technique*).
Typically a star scheme is made for optimising a single complex query; obviously I can have more than one star scheme for a single application, depending on the queries I must perform.



Pros of the Star Scheme

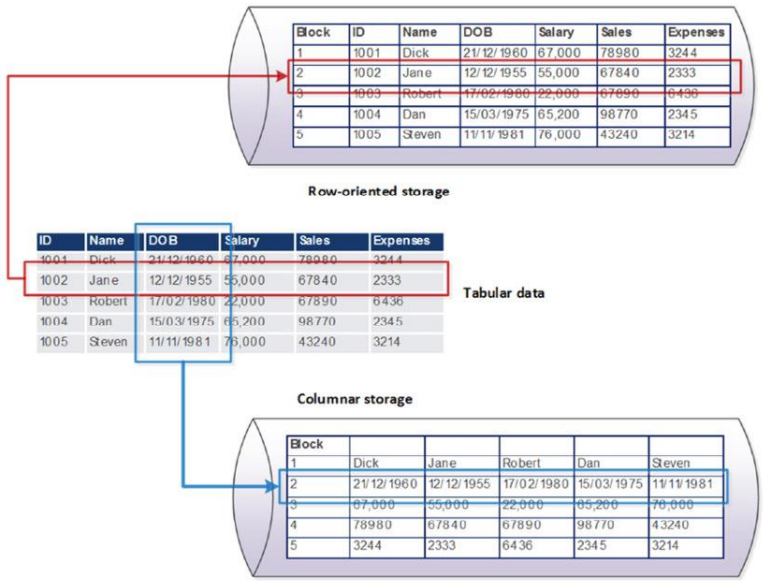
If we want to do some heavy analytics on sales, we can find all the information we need inside only one table, the central one.

Cons of the Star Scheme

- CPU and IO intensive.
  - The central table may become very big, so hard to manage.
- Data volume and the size of the central table grew up along user demands for interactive response times.

Column Storage

When we perform analytics, in most of the cases we only need the value of a single field of many rows. In a row storage this thing is possible but time-consuming, because I could do many accesses to the disk.



We store the set of the attributes of a table in the same section of the disk. In this way I can retrieve values of a certain column with much less disk accesses.

Pros of Column Storage

Queries that work across multiple rows are significantly accelerated.

Cons of Column Storage

Adding an entire record is complicated, because I must do many accesses to the disk.

Hybrid Solution

I got both solutions: A row-oriented for the OLTP side (*to have fast write operations*) and a columnar one for the OLAP side (*to have fast analytics*). Then, eventually the OLAP side will be updated with the new information gathered by the OLTP side.

Example of columnar vs row-oriented

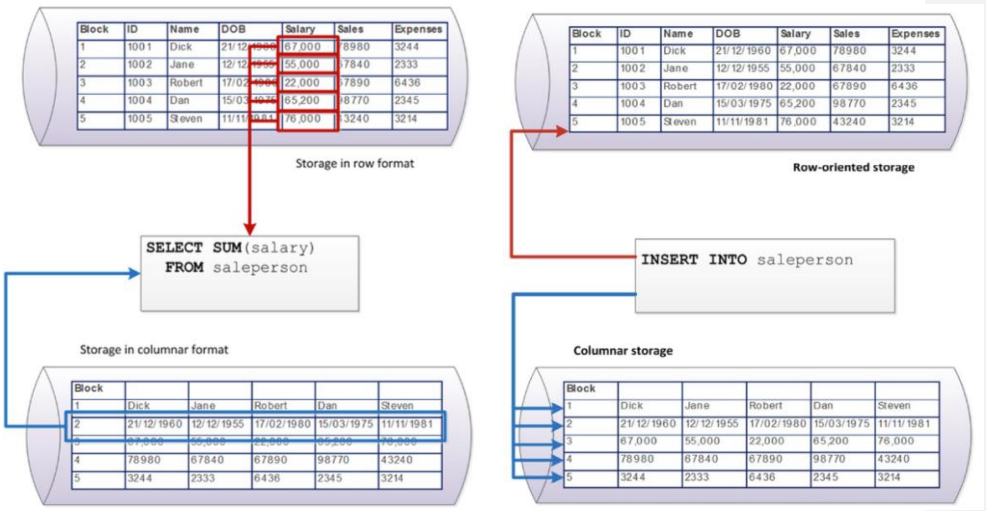
Let's suppose we got a RDBMS and we want to do some analytics (*total sum*) about the salary of employees.

Row storage

For perform the analytic we must access all the rows, so we have to do multiple accesses to the disk. To perform an insert operation of an entire record we have to access only one disk block.

Columnar storage

For perform the analytic we must make access to only one block of the disk.
All single row operations performance drastically decreases, so to perform an insert operation of an entire record we may have to access multiple disk blocks.



Columnar Compression

Data compression algorithms basically remove redundancy within data values. The higher the redundancies number, the higher the compression ratios.

How to find redundancies?

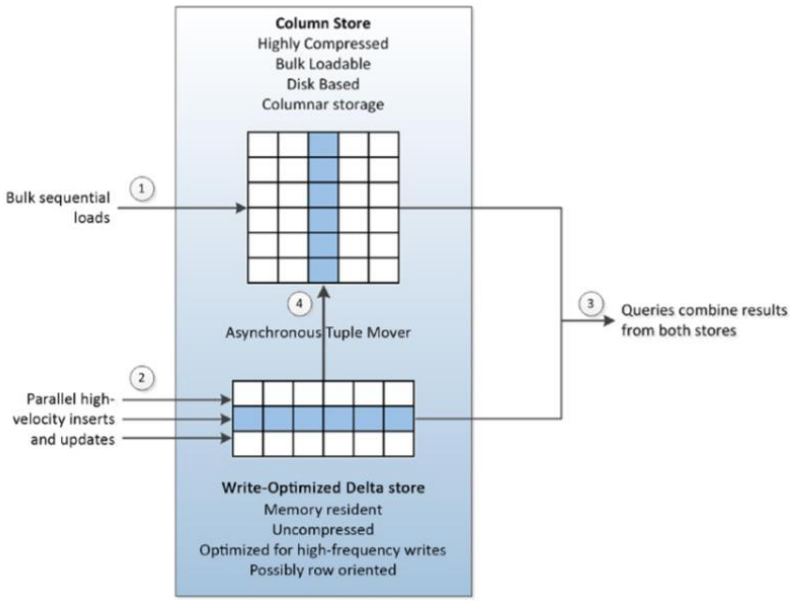
Data compressions algorithms usually try to work on localised subsets of the data. If data is stored in a column database, a very high compression ratio can be achieved with very low computational overheads.
As an example, often in a columnar database data are stored in a sorted order.
In this case, very high compression ratios can be achieved simply by representing each column value as a "delta" from the preceding column value.

Delta Store

The basic column storage architecture is unable to cope with a constant stream of row-level modifications. But these modifications are to be done instantly only if the data warehouse must provide real-time "*up-to-the-minute*" information.
The delta store is an additional part of the database (*often row-oriented, so it can optimise insert operations and row-level read operations*), devoted to merging the information of the last write operations to the database. Here we don't have compression and it is designed to optimise high-frequency data modification operations. In case of analytics that require real-time results, we may need to access both the delta store and the column store to return complete and accurate results.

How does the Delta store work?

1. When a write operation occurs, generally new data must be inserted.
2. Those new data stay temporarily in the delta store (*they don't go to the columnar storage for now*).
3. Then during a low load period, data will be shifted from the delta store to the columnar one.

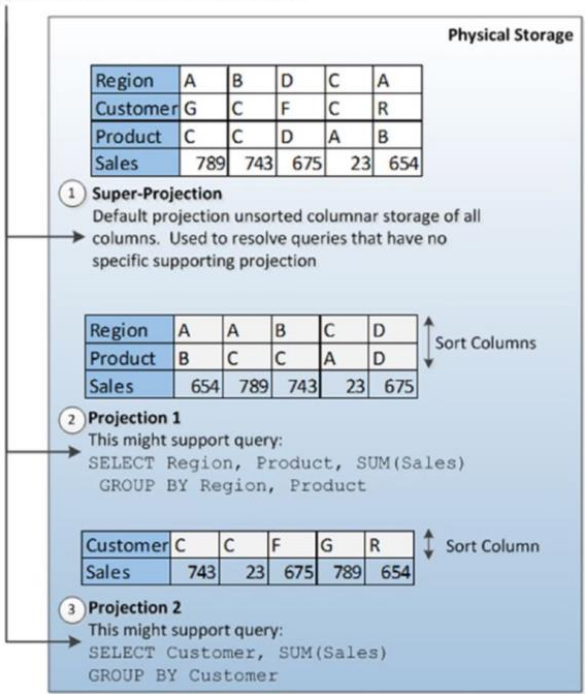


Projection

Sometimes, queries need to read more than one column (*we have to use a subset of attributes together*). A projection is a group of columns that are frequently accessed together. Columns of specific projection should be stored together on the disk (*so retrieving all of them is much less time-consuming*).
An important one is the Super-projection, which is the projection of all the columns of the database. Some examples of projections can be seen in the image below:

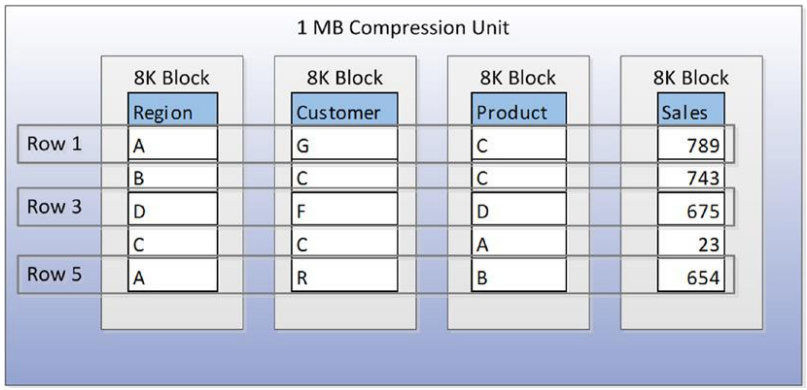
Region	Customer	Product	Sales
A	G	C	789
B	C	C	743
D	F	D	675
C	C	A	23
A	R	B	654

Logical Table
Table appears to user in relational normal form



Hybrid Columnar Compression (Oracle)

Data is divided into main units of 1MB.
Each unit is divided into blocks of 8KB.
Each block contains the values of a certain column.

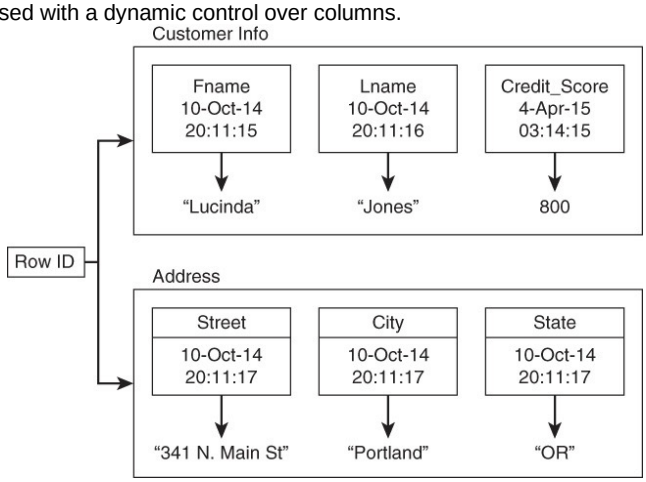


Key-Value vs Document vs Columnar Databases

- Key-Value databases lack support for organising elements with many attributes and keeping frequently used data together.
- Document Databases does not have a SQL-like query language.
- Document Databases do not make control on the data
- Columnar databases lack performance regards write operations.

BigTable – The Google NoSQL database

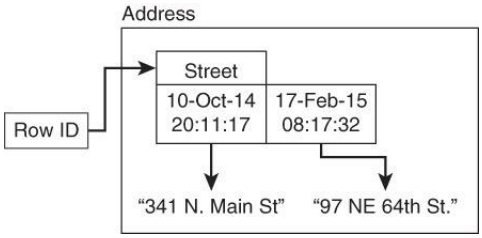
Is column-based with a dynamic control over columns.



A certain value is indexed by a row identifier, column name and a timestamp:

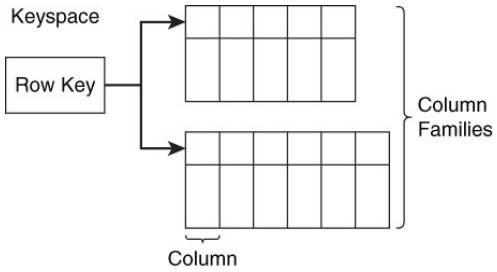
- With the `Row ID` I can access the entire row.
- With the `column name` I can access the value.
- With the `timestamp` I can access older versions of the value.

An important property of BigTable: Every read or write operation on a row is atomic, so I can't access two rows simultaneously.



Keyspace

The Keyspace is a top-level logical container: is the set of all possible keys that can be used to access data in the BigTable database. Typically, there is one Keyspace per application.



Column Families

A single data record is a row. A row is composed of several column families. Each family consists of a set of related columns.
The point of strength of BigTable is:columns that are frequently used together are in the same column family and column values within a column family are kept together on the disk.

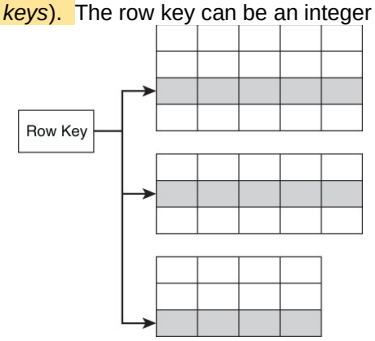
Street	City	State	Province	Zip	Postal Code	Country
178 Main St.	Boise	ID		83701		U.S.
89 Woodridge	Baltimore	MD		21218		U.S.
293 Archer St.	Ottawa		ON		K1A 2C5	Canada
8713 Alberta DR	Vancouver		BC		VSK 0AI	Canada

How to modify the data?

Column families must be defined a priori (*like the tables in relational databases*). Columns in a family can be modified dynamically, so applications can add or remove columns inside families (*like KVDB*).

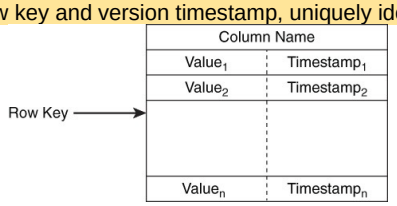
The Row key

The row identifier (*or row key*) is a sort of primary key that regards a piece of information (*not a single entity, like primary keys*). The row key can be an integer or a string.



Columns

A column, along with a row key and version timestamp, uniquely identifies values.



Column Database Architecture

How to build replicas and shard for columnar databases. The modern columnar databases are designed for ensuring high availability and scalability, along with different consistency levels.

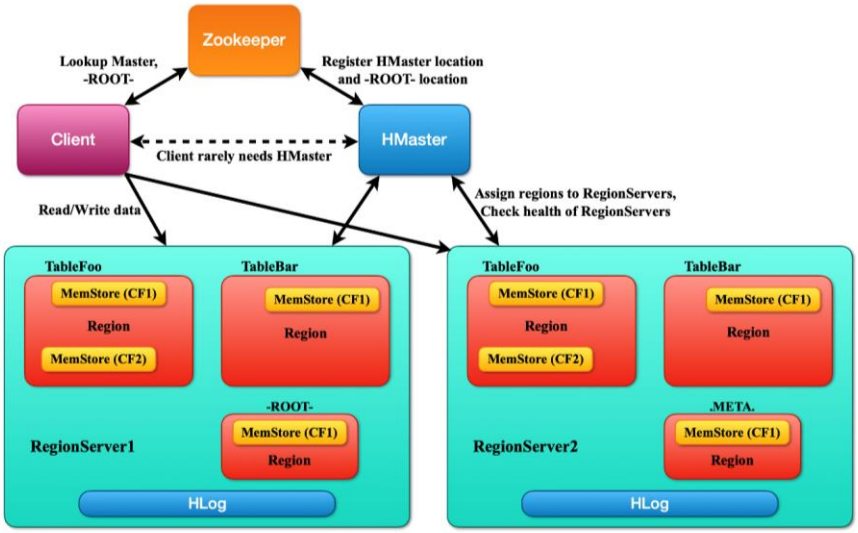
How to divide in partition a columnar database

A partition is a logical subset of a database, usually a particular column or column family.
Or a partition can store:

- A range of values Row ID or Column Name.
- Hash value based Hash value of the column name.
- List of values Name of states or provinces.

Master (HBase)

It is the Hadoop database, so a distributed, scalable big data store.



The file system permits us to see data split on various servers like one unique node.The file system of HBase is called HDFS.
The master has the duty of associating to the region server a specific portion of the data. The region servers are the end nodes that manage the user request.
The zookeeper is a node of Hadoop that coordinates the communication between the master and all other nodes.

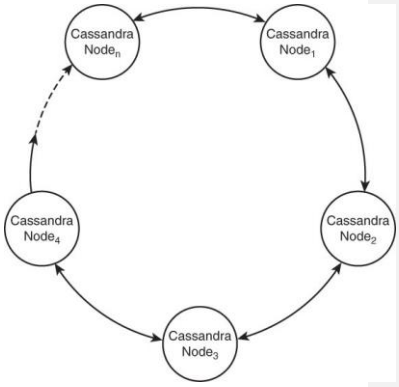
Cons of HBase

- We must deploy the software for the master, region servers and for the Zookeeper.
- The zookeeper and the master are both single points of failure.

HBase is based on the distributed file system of Google, the data is distributed in different servers, we have data partitioning, data redundancy and so on.
Name node contains the collections of nodes, the data node contains the collections of data.

Masterless (Cassandra)

In Cassandra, the first thing is that all servers have the same software installed. Nodes can still have specific functionality, but in general they are all the same, so they have the same capabilities.
Servers adopt a strategy to communicate with each other.



How is availability assured?

No point of failures, when a node fails, the entire system can still respond to the user, another server will respond to the operation.

Commit Log

Data structure for speed-up write operations.
A commit is a conclusion of a transaction, in this case we are sure we have respected the atomicity.
The commit log is an append-only file (*no delete, only append to the end)*.

1. The user sends a write operation to a random server.
2. The server will respond to the transaction and write it in the commit log.
3. Someone in charge of reading the info on the commit log, will update the information on the disk (*so will effectively send the write operation*).

Speed up write operations

The server can avoid waiting until the write operation has been completed. The write operation will be stored in the commit log, then, will be effectively sent to the target server to be performed.

Update propagation

The commit log file can be sended to the other servers.

Recovery

In case of disaster, we can recover the portion of the database by using the commit log.
All the write operation will be executed a second time (*keeping the original order*), and at the end the database partition will be recovered.

Bloom Filter

Probabilistic data structure used to speed up read operations.
The best way to explain this is with an example:

1. We want to retrieve information about Pisa in the database.
2. I don't know where it is (*neither if it exists or not*), so I must check if that value exists in the database.
    - I have to scan the entire partition.
3. If I do not find it, I may have to repeat the operation for another partition.

At the end, we performed more than one disk access.

The bloom filter evaluates whether an element is a member of a set or not. The answer given by the bloom filter is: "*Yes*"(actually "*Maybe*") or "*No*" .

I cannot have a false-negative, but I can obtain a false-positive.

If the answer is no, I am sure that in that set the info does not exist in that set.

Member Function

Input Set	Test Element	In Set
{a,b,c}	a	Yes
{a,b,c}	c	Yes
{a,b,c}	e	No

Bloom Filter

Input Set	Test Element	In Set
{a,b,c}	a	Yes
{a,b,c}	c	Yes
{a,b,c}	e	Yes
{a,b,c}	f	No
{a,b,c}	g	No

Low Probability but Possible

How to create and how to use a bloom filter?

The Bloom Filter is an array of `n` positions initialised to `0` at the beginning.
Then we can select `k << n` hash functions which map each data to be stored into the partition in an element of the array.
The hash function maps the value in a position of the array.
I do the operation (*explained below*) for every element inside the database (*so creating a bloom filter is time-consuming, but you do this rarely*).

When a new data `x` is added (*write*) to the partition:

1. We calculate the `k` hash functions.
2. Then we apply the `modulo n` to each of them.
3. The modulo gives us a number between `0` and `n-1`, so it can be used like an index of the array.
    - The module gives us k positions of the array (*some of them may be equal*).
4. We put `1` inside the `k` position of the array calculated by the `k` hash functions.

When a read operation on `x` occurs:

1. We calculate the `k` hash functions.
2. The access to the partition is performed if and only if all the corresponding element of the array are set to `1`.
    - If even one of the K positions is not equal to the obtained result, then I am sure that x is not in this partition.

Bloom filter - Explained in the details

The array of the bloom filter is unique for the partition.

So, if an element of the array is 1, then it means that during the insert of an element the location has been set to 1.
But if the location is 0, then I am sure that any of the elements inserted into the partition set the location to 1.
So, if the 2° location is set to 0 and the element that I am searching has `h2(x) = 1`, then I am sure that that element is not in that partition.
But if the 2° location is 1, then I am not sure if `x` is in the partition or not, because another element `y` could have set the location to 1.
The maximum false positive rate depends on `n` and `k`. The probability of a false-positive decrease if `n` increase. Obviously if n increases, the probability that the k hash functions set the same bit of two values, x and y for example, is lower.

Suggested value of `n` and `k`:

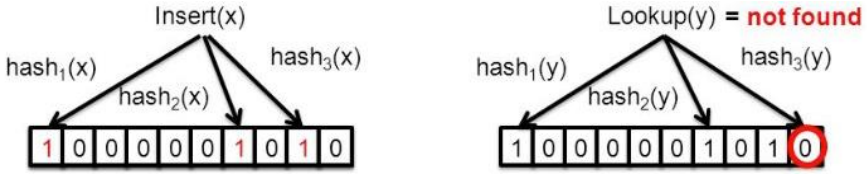
- `n` ∝ `O(h)`
- `k` ∝ `Log2(n/h)`

A bloom filter can be used only if I cannot remove elements from the data structure!

If I want to modify it, I must delete it and rebuild it from scratch.

Example of bloom filters

If we have `n=10` and `k=3`, whenever we make the insert, we will activate the hash function.



Why can't I remove elements?

Because if we remove some information, and put `0` in the position of the array, we will generate false negatives (*because some other elements set that location to 1*).
That for the definition of the bloom filter cannot occur.

Consistency Level

The consistency level refers to the consistency between copies of data on different replicas. The chosen consistency level depends on the specific application requirements. We can consider different levels of consistency.

Strict Consistency

I always want the query to return the most recent value; so updated data must be written in all the replicas.

Low Consistency

Data must be written in at least one replica. Some replicas could store old values and queries can return not updated values.

Moderate Consistency

All the intermediate solutions.

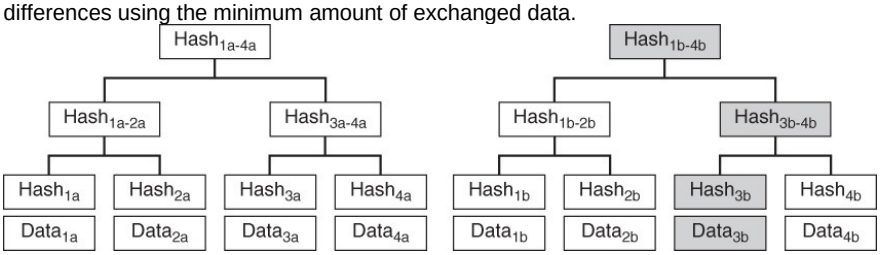
Replication

The main problem is to know when and how to update the replicas. Usually there is a main replica (*the first*) which oversees identifying the position of other replicas among the available servers. The primary server is often identified by a hash function.

Each database has its own replication system.

Anti-Entropy

Is a strategy (*that can be used in all kinds of databases, but heavily used in columnar databases*) for detecting differences (*inconsistencies*) in replicas. It is important to spot differences using the minimum amount of exchanged data.



Every server builds its own hash tree. Servers exchange hash trees among them to spot differences. Each leaf contains the hash value of a data set in the partition. Each parent node contains the hash value of the hashes of the child nodes.

The philosophy is: *Confronting a hash value is much faster than scanning the entire partition.*

1. A server receives the hash tree of another node and confronts it with its hash tree.
2. If the two roots are the same, then the copies are identical, and the process ends.
3. Otherwise, it will navigate the tree and find the different partitions.
4. The server with the older partition gets updated by the other.

Communication Protocols

We need protocols to check the status of the other servers.

Fully Connected

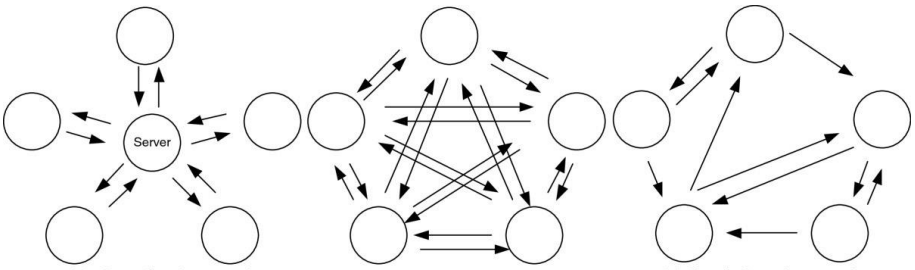
The simplest way to communicate is that every server communicates with all servers, but if there are a lot of servers we cannot use this approach because the number of messages increases exponentially with the number of nodes.

Centralised Approach

The central server collects info from other servers and then updates the others.The central is a single point of failure.

Gossip Protocol

Each node sends info about himself to a certain number of random servers. Can be demonstrate can the system converge in a very fast way.



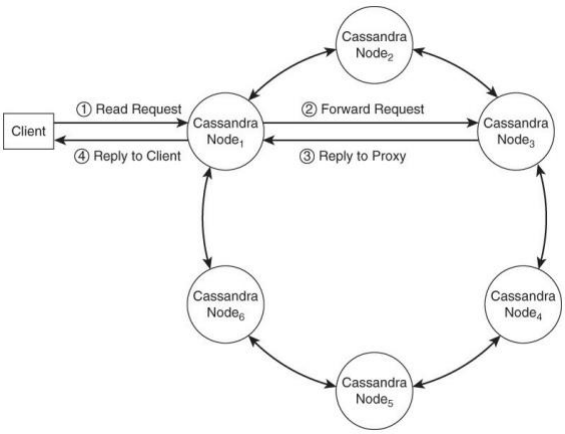
(a) Centralized approach, which uses a centralized server for forwarding and mixing of multimedia streams

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

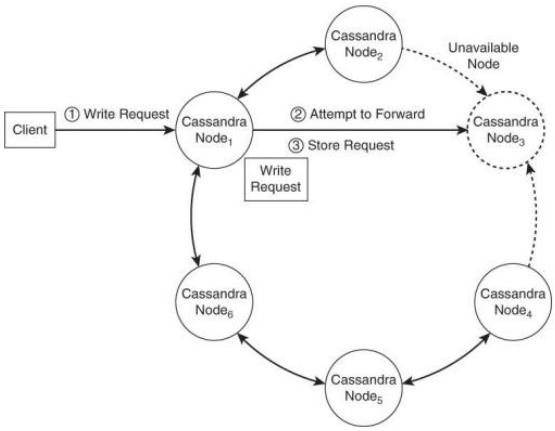
(c) Gossip-based approach, where peers operate in parallel, and each peer communicates with one or more randomly selected partner

Cassandra Approach

Any node can answer to the client. The request can be executed by that server, or if it is having an overload, the request will be forwarded to another server.
The first server will function as a proxy; the second server can respond to the read operation or (*like the first server did with him*) forward the request to a third server and act like a proxy too.



In case of a write request: the first server forward the write operation to the server where the write should be performed.



Hinted Handoff Process

Often asked in the exam.
What if the node interested in a write operation is down?
Two scenarios may occurs:

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

The hinted handoff process oversees:

1. The first server checks if the target node is on.
2. If it is off, the first server sends the write info to the designated node.
3. The designated node creates a simple data structure (*like a commit log file*) to store information about the write operation and where it should be performed.
4. The designated node periodically checks the status of the target server.
5. The designated node will send the write operation stored in the log when the target is available.

Exercise – Master Program

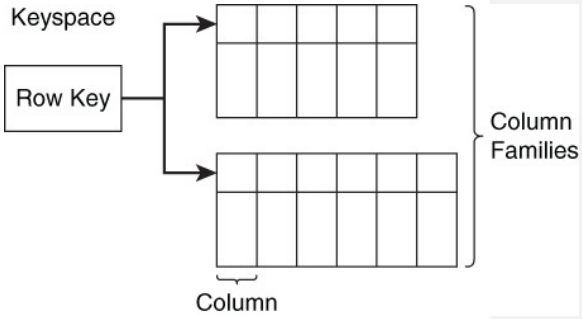
At the time of the exam, the student already saw how to use MongoDB, so they know that there are some functionalities of MongoDB for doing analytics.

Students often make errors.
I must reduce the number of accesses to the database; I don't care I have some redundancies.
In the exercise, this was three times in the exam, especially the second question.
The student can access his/her home page, and he/she must access to the list of my exams, he/she also see the exams that he/she passed, and then we want to make a query to see for each course the average mark.
One thing is the side of the application, data structures and variables, another thing is the database, we must reduce the access to the database, this is a schema-less database. We can specify the attributes that the user wants to retrieve it is called projection, thus just the needed information.

Exam questions: describe the cap theory, what is the most relevant feature of key value database with examples.

Columnar databases – Design tips

In this type of databases, we can store information in a strong unnormalized form, so, we must define the so-called key space, which is composed of a set of rows keys.



For each table we have a row key with a specific meaning, for each row key we have the column families which contain a set of attributes that have some common characteristics.

If we want to define a new column family, we must add a new column with a new list of attributes.
We don't have a scheme; we must know in advance which attributes must be used together.

Typical queries in columnar databases

- How many orders have been placed in the X region?
- When did a particular customer last place an order?
- What orders are in route to customers in London?
- What products in Ohio warehouses have fewer than the stock keeping minimum number of items?

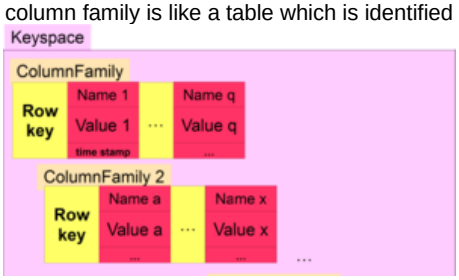
Information needed by the Database design

Queries provide information to effectively design column family databases.
The information includes:

- **Entities.**
  - A single row describes the instance of a single entity.
  - Rows are uniquely identified by row keys.
- **Attributes of entities.**
  - Attributes of entities are modelled using columns.
- **Query criteria.**
  - The selection criteria should be used to determine optimal ways to organize data with tables and column families and how to build partition.
- **Derived values.**
  - It is an indication that additional attributes may be needed to store derived data.

Differences between Relational and Columnar databases

Inside the keyspace, a columnar database stores the column families. For simplicity, you can think of the keyspace like a RDBMS schema and column families like tables, but their similarities end here. A column family is like a table which is identified by a row key.



Column families "float around"

A column family "floats around", it is not linked to any other column family (*in the relational one, tables can be linked with relationships*).

Column families do not represent an entity

A column family consists of a set of data about a specific topic, a column family does not represent an entity (*in the relational one, tables represent entities*), in fact data presented in a column family could have been taken from multiple rows.

Column families do not have a schema

Columns can vary between rows (*see the image below*): in a column family, rows can contain a different number of columns to each other, as well as data from different columns.
Moreover, in a column family, columns can be added dynamically at different times (*instead, in a RDBMS you can't easily change the structure of a table).*

Join operations can not be made easily

Join operations can't be used because data is denormalized.

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

Empty values

In a relational database a table is always a complete matrix: if we don't have a value for an attribute, we set it to `null` and at a low level will occupy some memory.

Let's consider 3 columns families {`1` , `2` and `3`} and the row with `key = "Pietro"`.

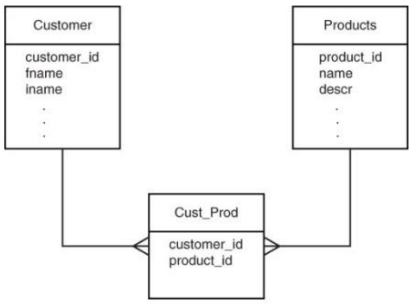
- 1° column → `2`, `3`, `9`.
- 2° column → `9`, `1`, `1`.
- 3° column → empty.

In a columnar database `Pietro` is composed in this way:
`Pietro{{2, 3, 9}, {9, 1, 1}}`
The third column (*which is totally empty*) doesn't occupy any space, so in the columnar database an empty information does not occupy memory.

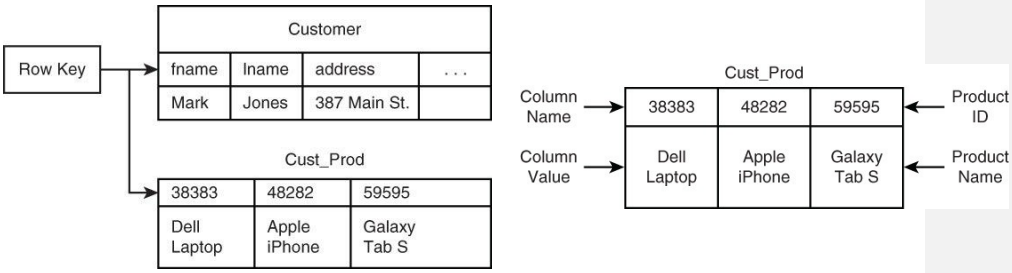
In a relational database the same information would be stored into a vector:
`[2, 3, 9, -1, -1, 9, 1, 1, null, null, null]`
It occupies more memory than the columnar one, because the length of the structure doesn't change even if some elements are empty.

Avoid join operation

Sometimes we need to do join operations, but we only care about the result so to avoid doing join operations, I add a data structure with derived values that I normally (*in a RDBMS*) could obtain only with a join operation.

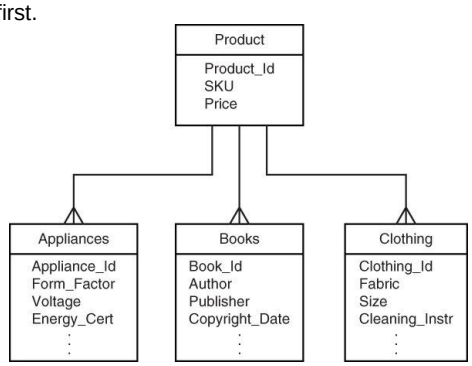


For every customer I create a redundant structure which contains the list of the object IDs. The redundant structure is a column family, where every column is a value itself (*as can be seen in the image below at the right, the name of the column can store information too*). This column family is linked to the same row key of the customer column family, so using the key `customer1` we can also access the column family of all the products ordered by that customer.



Model an entity with specialisations

Suppose that we have this situation, we have an ecommerce application, we have a product, and this is specialised in three different products.
In a relational database we may create 4 tables, where one is the father and the other 3 are specialisations of the first.

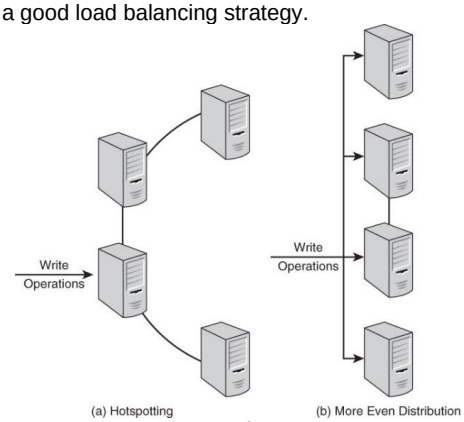


To have an equivalent representation in a column database we need just one table with four columns.The information will be represented like in this case:

If we can easily predict the number of products, we can follow this approach. If the number of products change too often, then this approach is not the right choice.

Hotspotting and Load Balancing

When only a part of the cluster works. This may happen when the work is not distributed properly, so we need a good load balancing strategy.



We can use a sequential Hash function (*at the first operation give 1, at the seconds give 2…*) that takes as input the row key value. In this way the write load will be equally distributed among the servers.

How to manage the previous versions of values

BigTable permits us to keep old versions of the values and they are useful in case of roll back changes. Mind that those values will occupy memory and keep them even if they are useless is a waste of memory. The number of versions that you have to keep depends on the requirements of the application.

Avoid complex structures

If the structure to store is too complex (*because every attribute needs to be separate in different columns*) it may be a good idea to simply store the JSON version of it like a big string. But this trick have some backfire too:

No Indexes

Using separate columns for each attribute makes it easier to apply database features to the attributes (*indexing*), so if we simply store the JSOn string we can't use indexes anymore.

We don't have any column families-related strategy

Separating attributes into individual columns allows you to use different column families if needed.

Indexes

In a column database an index is used to quickly find rows that references that column value.
We have two types of indexes:

- Primary indexes are indexes on the row keys.
- Secondary indexes are indexes created on one or more column values.

If we want to speed-up a query like this:

```
SELECT * FROM Tab WHERE state = "OR" AND Lname = "TR"
```

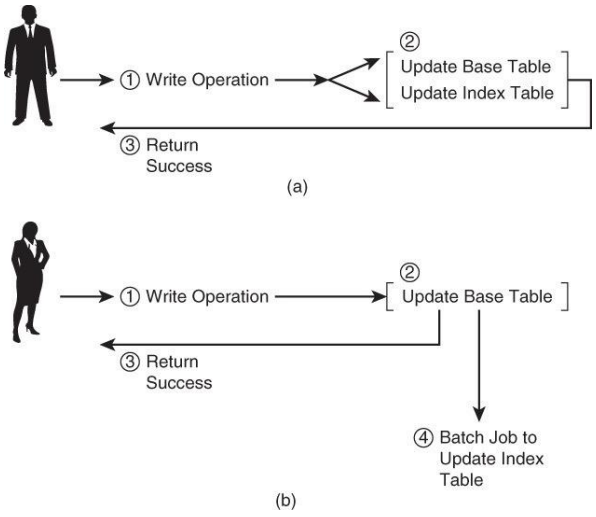
Then we should put a secondary index on `state` and `Lname`.

When an index is useless?

- When the distribution of the values in a column is uniform (*roughly equal numbers of each value*).
- When the distribution of the values in a column is composed of only two values (*like a boolean*).
- When the number of different values is too high.
- When in a column we got too many empty values.

How to maintain indexes

The first is: Updating by our application an index table during the write operation and then returning the confirmation to the user only after the index has been updated.
The second is: Batch more than one updates of the index and perform them in a second moment all in one time, the application is faster but we leave indexes temporarily not updated.



Last Update: Thursday, 22 February 2024

Graph Databases

Recap of graph's theory

Vertices

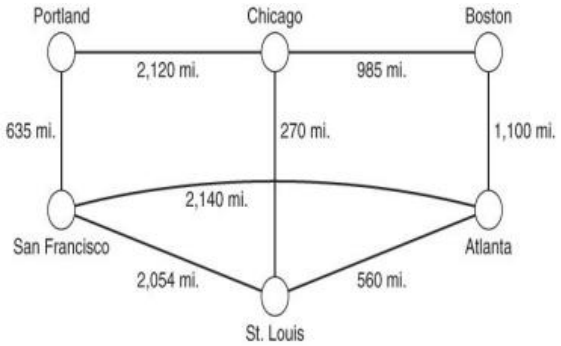
In graph theory we use a vertex for representing an instance of an entity. Vertices (*obviously*) have some interesting properties that we'll exploit later.

Edges

Edges (*also called links*) represent relationships between two vertices (*entities*). An edge can have properties such as the "*weight*" (*the weight commonly represents the "cost" to traverse that edge*).

Example of a Graph

In the example below, vertices represent cities of the United States and the edges represent the roads that connect two cities.



Properties that a vertex may have: population, geographic coordinates, name of the city (*like in the image*), geographic region, and so on. Instead, the properties that an edge can have may be: The distance of the road that links two cities (*like in the image*), the speed limit of the road or the number of lines of the road.

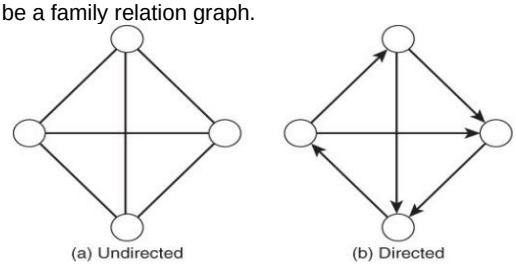
Types of Graphs

Undirected graph

The edges do not have a direction. An Example of this type of graph may be highways network graphs.

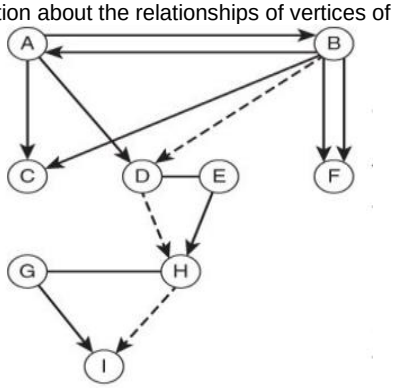
Direct graph

The edges do have a direction. Some edges may be unidirectional, or the edge that goes from X to Y may have different properties of the edge that goes from Y to X. An example of this type of graph may be a family relation graph.



Path in a graph

A path is a sequence of nodes and edges, that from a node `X` take us to another node `Y`. Paths allow us to capture information about the relationships of vertices of a graph.



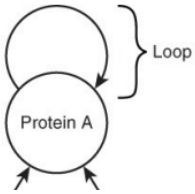
In the image above, we can see the dotted path: `B → D → H → I`

Dijkstra Algorithm

Find (*among all possible paths*) the path starting from a node `X` and ending to a node `Y` with the minimum cost.

Loops

Is a type of relationship between an instance of an entity and itself. An example of this type of relationship may be the proteins, a protein of a certain type may interact in a certain way with proteins of the same type.



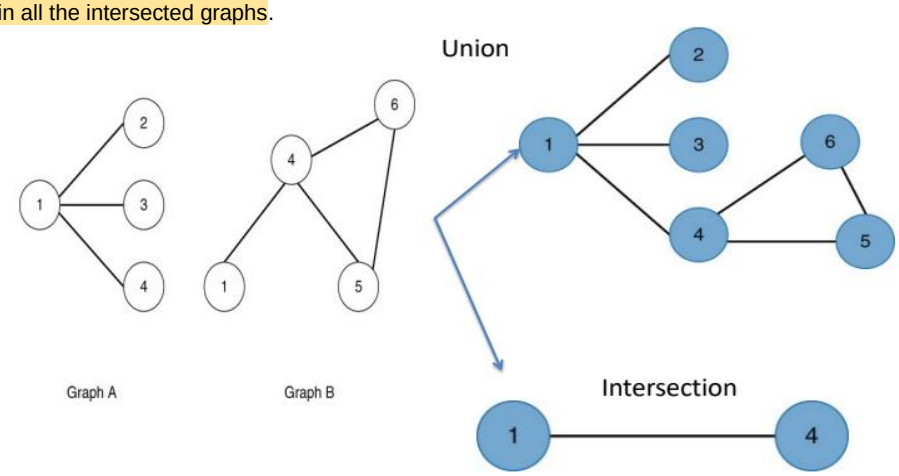
Union & Intersection of Graphs

Union

The union of graphs is a graph composed by all nodes and edges of the united graphs. Nodes and edges in common are not repeated.

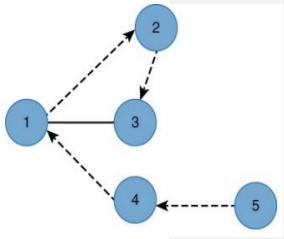
Intersection

The intersection of graphs is a graph composed by all nodes and edges that are in common in all the intersected graphs.



Graph traversal

A graph traversal is the process of visiting all vertices in a graph, often to set or read some property value in a graph. It's common that, in a graph traversal, a vertex is visited more than one time, so a graph traversal could have a non-linear complexity. Graph traversal can be a type of operation that you may tend to do very often, so you shouldn't choose a graph database in an application where you will end up doing many queries that need to explore all nodes of the graph; Instead should be exploited for applications that require exploring relationships.

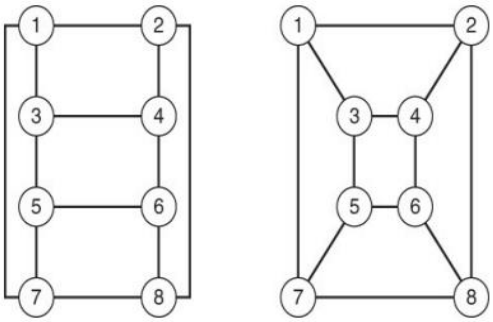


Isomorphism

Two graphs (*or subgraphs*) are isomorphic if:

- For each vertex in the first graph there is a corresponding vertex in the second graph.
- For each edge between a pair of vertices in the first graph there is a corresponding edge between the corresponding vertices of the second graph.

Detecting this characteristic is useful for exploiting useful patterns among the graphs.



Properties of graphs

These two properties are related to the complexity of graph traversals and of the memory occupancy.

Order

The order is the number of total vertices in the graph.

Size

The size is the number of total edges in the graph.

Properties of a vertex

Degree

The number of edges linked to the vertex: the higher the degree is and more "*important*" the vertex is in the graph.

Closeness

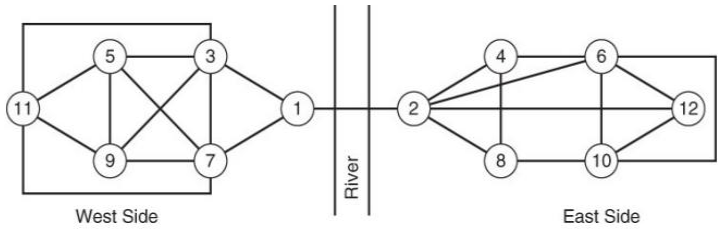
How far the vertex is from all the other vertices in the graph. Given a vertex `X`, the closeness can be calculated as:

$$C_B(x) = \frac{1}{\sum_y d(y,x)}$$

Where the denominator is the sum of the length of the shortest paths between the node `X` and all other nodes `Y` of the graph. The closeness is useful to evaluate the speed of information spreading in a network, starting from a specific node `X`.

Betweenness

Counts the number of times a vertex appears on the shortest path between any two other vertices, so as to represent the significance of this vertex to the network connectivity. It can be seen as the level of how much a vertex **can be** a bottleneck. (*so if the vertex has bad performances, then is more probably that the vertex can become a bottleneck*).

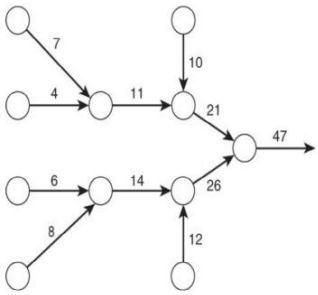


In the example above, the vertices 1 and 2 will surely have a high betweenness, because they are the only one to provide a link to the other side of the river.

Other types of graphs

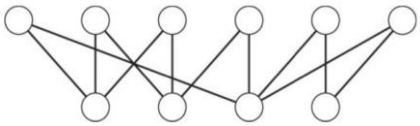
Flow Networks

Is a direct graph with capacitated edges used in logistics. In this type of graph each vertex has a set of incoming and outcoming edges and the sum of the capacities of incoming edges cannot be greater (*can only be equal or lesser*) than the sum of the capacities of outcoming edges (*I can't send more than I am receiving*). The unique exceptions to this rule are the sources and the sink nodes.



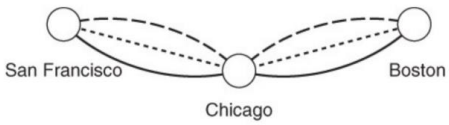
Bipartite graph

Graph used for application of courses and students. It can be used for modelling relationships between students and teachers. We have two sets of vertices and each vertex can be connected only to the vertex of the other set.



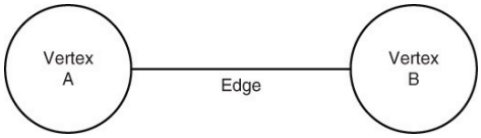
Multigraph

We can have multiple edges between two vertices. It is useful when we have multiple ways to reach another node (*between two cities there can be more than one road*). Note that given some edge that connect the same two nodes, each edge can have different properties than the other edges.



Graph databases introduction

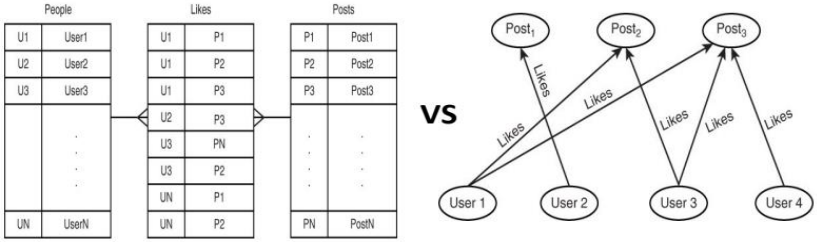
A vertex represents an entity, and usually, entities in a graph belong to the same category. An edge specifies a relation between two vertices, and relations may be "*long terms*" or "*short terms*".



Graph databases vs Relational databases

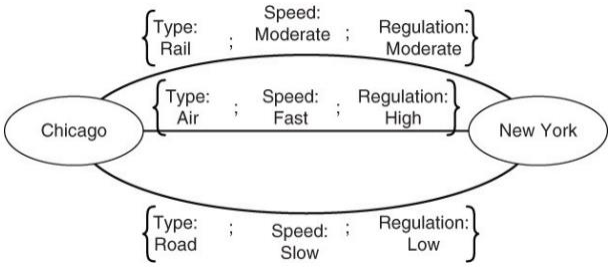
Query: List all courses where a particular student is enrolled in.

In a RDBMS we would need to do a join operation between at least two tables. Instead in a graph database this query is faster because a relationship (*edges*) between two entities (*vertices*) are explicitly stored, so (*like in the example below*) in a bipartite graph we only need to check if between a student and a course there is an edge.



Relationships in graph databases

Relationships may have properties too, and in a graph database this thing is easy to implement, because I can have multiple types of edges (*specifying different values of edge properties*) and each type of edge can have different properties..



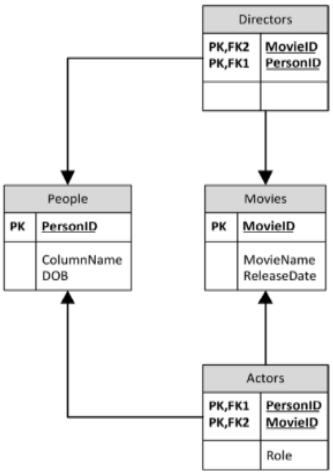
Typical queries on a graph database

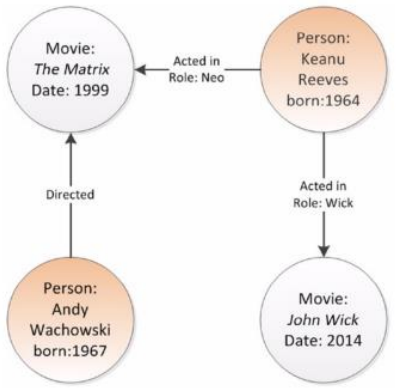
- How many edges between two nodes with certain properties?
- How many edges are linked to a node.
- Closeness and betweenness of a node?
- Is a node a bottleneck?
- If a node is removed which parts of the graph become disconnected?

Example of graph database: HollywoodDB

Let's design a DB for handling movies, actors and directors. Let suppose to offer the following services for users:

- Find all the actors that have ever appeared as a co-star in a movie starring a specific actor.
- Find all movies and actors directed by a specific director.
- Find all movies in which specific actor has starred in, together with all co-stars and directors





SQL lacks the syntax to easily perform a graph traversal, especially traversals where the depth is unknown or unbounded. Performance degrades quickly as we traverse the graph. Each level of traversal adds significant response time to a specific query. In the right image we can see the equivalent graph model.

Last Update: Thursday, 22 February 2024

Example of graph database: Social Network

"A social network like stack overflow"

cit. Prof. Pietro Ducange

Define the main use cases

1. Join and leave the site.
2. Follow a developer.
3. Post questions for experts in a specific area.
4. Suggest new connections with other developers based on shared interests.
5. Rank members according to their number of connections, posts and answers.

Define entities and their properties

Developer	Post

• Name
• Location
• NoSQL databases used
• Years of experience
• Area of interests
	• Date of creation
	• Topic Keywords
	• Post type
	  ○ Example
	  ○ Question
	  ○ Tip
	  ○ News
	  ○ …
	• Title of the post
	• Body of the post

Identify relationships between entities

1. Developer Developer.
2. Developer Post.
3. Post Developer.
4. Post Post.

Let's explore the first relationship. A developer can follow another developer, so the name of the first relationship could be "*follows*". If a developer X follows another developer Y, then X is interested in Y's posts.
Let's explore the second (*and the third*) relationship. A developer can create and publish a post, so the name of the second relationship could be "*create*". At the same time, another relationship is created (*one of the third type*), because at the same time the post is created and published by the developer. So, the name of the third relationship could be "*is created*".
Let's explore the fourth relationship. A post can be a reply to a question (*which is a post itself*), so the post that contains the reply is linked to the post which contains the question. The name of the fourth relationship could be "*refers to*". The two posts can be created by two different developers in different moments (*even with a difference of years between the creation dates*).

Mapping queries

Remember: Queries drive the design of the graph database.

Domain-specific queries must be translated into graph-centric queries. But only after the identification of entities and relationships.

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?

Implement graph queries

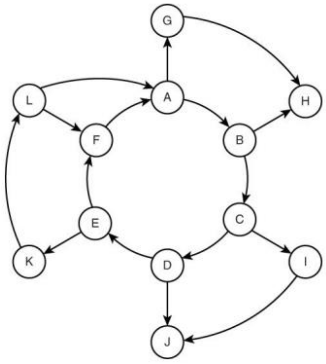
To implement them, you must use good graph algorithms.

Some advice

- Always control the dimension of the graph (*even if it is not easy*); bigger the graph, bigger the computations.
- Use indexes for improving retrieval times.
- Use appropriate types of edges.
  - If the relation is symmetrical, use an undirected edge.
- Design properly the properties of an edge.
  - Prefer to use simple edges because properties require memory.
  - Use the right types for properties.
    - If a property will have only the values `0` or `1`, use a boolean instead of an integer.
- To avoid endless cycles, put a time limit to every graph traversal.
  - Use proper data structures to keep track of visited nodes (*avoid reading the same node more than one time).*

Cycles in graphs

If there is no indication that we have already visited A, we could find us in an endless cycle of revisiting the same six nodes over and over again.



Scalability of graph databases

Most of the famous implementations of graph databases do not consider the deployment in a cluster of servers (*no partition nor replicas management*). Indeed, they are designed for running on a single server (*vertical scaling only*). For this reason, the developer must take special attention to the dimension of the database; in particular of: the number of nodes, the number of users and the number and the size (*type*) of properties.

MongoDB

MongoDB stands for "Hu**mongo**us **DB**" and in this course, we'll see some guidelines for working with MongoDB.

MongoDB key features

MongoDB is open-source and implemented in C++

We can use C++ for implement some analytics or procedures.

MongoDB is very well documented

Documentation contains a lot of examples.

MongoDB is document-based

We deal with BSON (*a JSON document with some additional properties*) format documents. The maximum dimension of a document is 16 MB.

MongoDB is organised in collections

Every document is stored in a collection. A collection is like a table of a RDBMS but documents inside a collection do not need to have uniform structures; they might share indexes (*which are implemented for a specific collection*).

MongoDB offers high performance and high availability

MongoDB gives support for embedded data models (*document embedding*), which reduces I/O activity on the database.

MongoDB offers a rich query language

MongoDB offers a query language (*like SQL, but only for MongoDB*) very easy to learn. This language supports CRUD (*read and write*) operations as well as: data aggregation, text search and support geospatial queries.

MongoDB offers support for replica management

MongoDB has a replication facility, called "replica set", which provides:

- Automatic failover
- Data redundancy

MongoDB offers support for Horizontal Scalability

MongoDB implements a system for creating managed shards.

MongoDB offers different configurations on CAP triangle

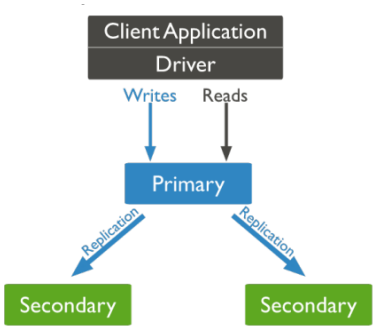
CP and AP mainly.

Configuration file and other files

MongoDB's configuration can be exploited via a configuration file: Windows `<install directory>/bin/mongod.cfg`

Architecture of MongoDB

Replication in MongoDB

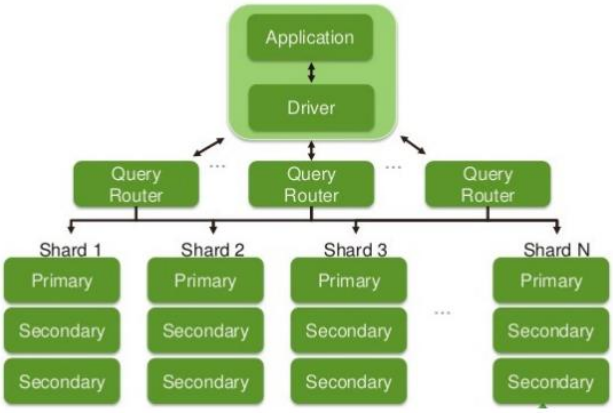


MongoDB replica management is Master-Slave (*primary-secondary*) based. A MongoDB instance is composed of all replicas and all shards.

1. The client application (*written in any programming language*) sends a request (*write or read operation*) to the primary (*master*) replica.
2. The master server oversees communicating (*forward*) to all (*or some of them*) slaves to update replicas.

Sharding in MongoDB

It is a bit more complicated than the replication process. Let's suppose that we want 2 shards; we need 2 primary servers for store shards. Each shard should be replicated at least 3 times; we need 6 secondary servers for replicas. We also need a server called "*Config Server*" (*which should be replicated too*); we need at least 1 more server. In total we need 9 servers to implement sharding with 2 shards.



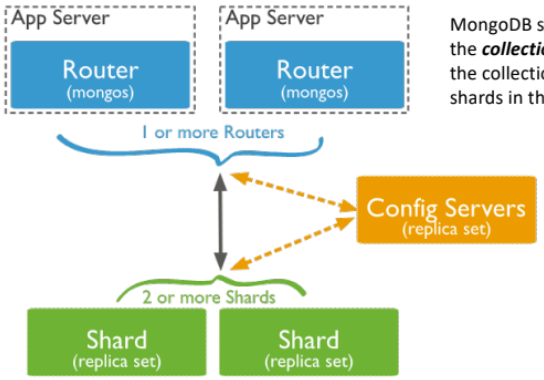
Core processes of MongoDB

`mongod`

The `mongod` is the daemon process which maintains the activity of the MongoDB server. It handles data requests, manages data access and database processes.

`mongos`

They are strictly related to sharding activities. The `mongos` is the controller and query router interfaced between the client application and the sharded cluster.



MongoDB shards data at the ***collection level***, distributing the collection data across the shards in the cluster.

Documents in MongoDB

The BSON file

BSON stands for "***B**inary J**SON***". It is the binary serialisation of JSON-like documents. Inside a BSON document we got additional data types and ordered fields. The BSON file is designed to be encoded and decoded in different languages.

```
{
     Name: "sue" ,
     Age: 26,
     Status: "A" ,
     Groups: [ "news" , "sports" ]
}
```

The syntax is key-value like → `<fieldName : value>`.

The `_id` field

By default, each collection contains an `*_id` field. This field is like a primary key: it is immutable, cannot be an array and must be unique for each collection. The default data type for this field is `ObjectId`, which is pretty small sized.

MongoDB vs SQL Database

mongoDB	SQL
Document	Tuple
Collection	Table/View
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

In SQL we joined operations to get normalised information from un-normalized information, in MongoDB we got embedding.

MongoDB Operations

Create & Insert operation

Create or insert a new document inside a collection.

Insert Operation

If the collection does not exist, the insert operation will create the collection too. Insert operation targets a single collection. All write operations are atomic on the level of a single document; so, a document is completely added or not added at all.

`insertOne()and insertMany()`

It takes a document-like parameter and inserts it into the target collection (`users`).

```
db.users.insertOne
({
     Name : “sue”,
     Age : 26
})
```

With `insertMany()` I can insert multiple documents in a target collection (*in the example the collection* `persons`).

```
db.users.insertMany
({
     Name : “Sue”,
     Age : 26
},{
     Name : “Alex”,
     Age : 25
})
```

Drop Operations

`drop()`

`db.users.drop()` it drops the target collection (`users`).

Query operations

Example of Queries

`find()`

Normally, the result set is sorted (*by default*) in ascending order by the `_id` field. Here some examples of queries:

```
db.users.find( {} )
```
"Get all the documents in the collection `users`*"*

```
db.users.find( { age : 26 } )
```
"Get all the documents in the collection `users` *with age = 26"*

```
db.users.find( { age : { $in: [ 25 , 26 ] } } )
```
"Get all the documents in the collection `users` *with age = 25* ***or*** *age = 26"*

```
db.users.find( { name : 'Alex' , age : { $lt : 25 }} )
```
"Get all the documents in the collection `users` *with name = 'Alex'* ***and*** *age < 25"*

The "And" condition is implemented with the **`','`**.

Instead, the inequality is implemented with `$lt` → *"Less Than".*

```
db.users.find( {
      $or[
            {name : 'Alex' },
            {age : { $lt : 25 }}
      ]
} )
```
"Get all the documents with name = 'Alex' ***or*** *age < 25"*

```
db.users.find({
      name : 'Alex',
      $or[
            {age : { $lt : 25 }},
            {age : { $gt : 26 }}
      ],
      job: /^p/
})
```
"Get all the documents with name = 'Alex' ***and*** *(age < 25* ***or*** *age > 26)* ***and*** *the job starts with a 'p'"*

Instead, the inequality is implemented with `$gt` → *"Greater Than".*

Instead, the "*starts with a p*" condition is evaluated through a regular expression.

Update operations

How to change something inside an existing document?

updateOne()

Update only one document, specified through filters.

```
db.collection.updateOne( <filters> , <update> , <options>)
db.items.updateOne
(
{ name : "Computer"}, // the item must be Computer
{
     $set: {"size.uom" : "cm" , status : "P"}
     // field to change
     $currentDate:{lastModified : true}
}
})
```

If in the document that we are updating the field to update does not exist, it will be created. If I am using `updateOne` and many documents are positive to the filtering, only the first that mongoDB finds will be updated (*generally the oldest one*).

updateMany()

Update many, instead, applies the update to all the documents that are positive to the filter.

Cursors

Cursors Keyword

Sort

Change the sorting method of the result set.

```
db.users.find({}).sort({ age : k })
```

ASC □ `k = 1` DESC □ `k = -1`

Example of Sort

```
db.inventory.find({}).sort({ year : 1 , qty : -1})
```

Ordered by year and for "*parimerito*" they are ordered for qty.

Commentato [1]: Check

Limit

```
db.users.find({}).limit(n)
```

`Limit(n)` only returns the first `n` rows.

Skip

```
db.users.find({}).skip(n)
```

`Skip(n)` ignore the first `n` rows.

Counting documents

```
db.inventory.countDocuments(query)
```

It does not return a cursor; it returns an integer, which represents the number of documents returned by the specified query.

Note that `countDocuments` is deprecated, but mongoDB will still accept this command. Please use instead:

```
db.inventory.find(query).count()
```

Queries on embedded and nested documents

How to perform queries with embedded or nested documents. We perform a filter like before, but now the "value" to test is a document. Let's suppose that the field size is a document:

```
Size { h : 4 , w : 21 , uom : "cm"}
```

So, we write something like (f*or strict equality*):

```
db.inventory.find { size : { h : 2 , w : 3 , uom : "cm"} }
```

Or something like this (*for inequality*):

```
db.inventory.find {
      "size.uom" : "cm",
      "size.h" : { $lt : 15 },
      "size.w" : { $gt : 15 }
}
```

We must quote the field too, in this way:

```
 < "field_name.attribute_name" : "condition" >
```

Queries on arrays

With square brackets

```
db.inventory.find { dim_cm : [ 14 , 15 ] }
```

The array must contain only those values and in that precise order. So, for the previous example:

- `[14 , 15]` will match the filter.
- `[14 , 16 , 15]` will **not** match the filter.
- `[15 , 14]` will **not** match the filter.

Without square brackets

```
db.inventory.find { dim_cm : 14 }
```

The array must contain that value. So, for the previous example:

- `[14 , 15]` will match the filter.
- `[14 , 16 , 15]` will match the filter.
- `[15 , 14]` will match the filter.
- `[15 , 16]` will not match the filter.

`$all`

```
db.inventory.find { dim_cm : { $all [ 14 , 15 ] } }
```

The array must contain (*among all its values*) those values without considering the order. So, for the previous example:

- `[14 , 15]` will match the filter.
- `[15 , 14]` will match the filter, the order does not count.
- `[14 , 16]` will **not** match the filter, because it does not contain `15`.
- `[14 , 16 , 15]` will match the filter.

Apply conditions

```
db.inventory.find { dim_cm : { $gt [25] } }
```

The array must contain at least one value greater than `25`. So, for the previous example:

- `[14 , 15]` will **not** match the filter, because neither `14` nor `15` is greater than `25`.
- `[15 , 30]` will match the filter, because `30` is greater than `25`.
- `[35 , 30]` will match the filter, because `30` and `35` are both greater than `25`.

Multiple conditions

```
db.inventory.find { dim_cm : { $gt : 15 , $lt : 20 } }
```

This one is quite complicated…The array must contain: At least one value greater than `15` **and** at least one value lower than `20` **or** at least one value greater than `15` **and** lower than `20`. So, for the previous example:

- `[14 , 16]` will match, because `14` is lower than `20` and `16` is greater than `15`.
- `[17 , 30]` will match.
- `[35 , 30]` will **not** match, because there aren't numbers lower than `20`.
- `[17 , 30]` will match.
- `[12 , 13]` will **not** match, because there aren't numbers greater than `15`.

"I do not know the utility of this thing"

Prof. Pietro Ducange

Projection of fields

```
db.inventory.find(
      {status : 'A'} , // Filter.
      { item : 1 , status : 1 , _id : 0} // Projection.
)
```

This query will show the `item` and the `status`, but not the `_id`. The `item : 1` specifies that I want to see that attribute in the result set of the query. The `_id : 0` specifies to suppress the `_id` field.

Pros of projection

Projection permits us to optimise the query on the client-side because more fields mean more bytes to send, so more latency and more bytes to manage.

> *"The projection must be used in the project… we will check this"*
>
> *Prof. Pietro Ducange*

Cons of projection

None, only a bit more work during the writing of the query.

Projection and embedded documents

```
db.inventory.find( {status : 'A'} ,{ item : 1 , "instock.qty" : 1})
```

The field `qty` is in the document embedded in the `instock` array.

```
> db.inventory.find(    { status: "A" },    { item: 1, status: 1, "size.uom": 1 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "size" : { "uom" : "cm" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "size" : { "uom" : "in" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "size" : { "uom" : "cm" } }
```

Projection and arrays

`$slice`

How to select specific elements of an array?

```
db.inventory.find( {status : 'A'} ,{ instock: { $slice : k } } )
```

With `k = -1` the query returns the `last` element of the array `instock`.
With `k = 1` the query returns the `first` element of the array `instock`.

Can I get the second or the third? Nope.

> *"It has been implemented in this way…Why? I don't know"*
>
> *Prof. Pietro Ducange*

`$elemMatch`

It returns the first value of a field of the array that matches a given condition. So in this case, the order inside the array may change the result of the query.

```
db.inventory.find(
      {status : 'A'} ,
      { instock: { $elemMatch : { qty : { $gt : 20} } } }
)
```

Queries on the type of the field value

Let's suppose that we got this collection:

```
Inventory
{
      { _id : 1 , item : null }
      { _id : 2 }
      { _id : 3 , item : 'book' }
}
```

$type

I can do queries based on the type of the field.

```
db.inventory.find( { item { $type : k }})
```

The number `k` is an integer which value represents a specific type of data, for example we got:

Type	Number	Alias
Double	1	"double"
String	2	"string"
Object	3	"object"
Array	4	"array"
Binary data	5	"binData"
Undefined	6	"undefined"
ObjectId	7	"objectId"
Boolean	8	"bool"
Date	9	"date"
Null	10	"null"

Regular Expression	11	"regex"
DBPointer	12	"dbPointer"
JavaScript	13	"javascript"
Symbol	14	"symbol"
JavaScript code with scope	15	"javascriptWithScope"
32-bit integer	16	"int"
Timestamp	17	"timestamp"
64-bit integer	18	"long"
Decimal128	19	"decimal"
Min key	-1	"minKey"
Max key	127	"maxKey"

Returning to the collection above if `k = 2` (`string`) then the query:

- The `_id = 1` will not be shown.
- The `_id = 2` will not be shown.
- The `_id = 3` will be shown.

About the `null` type

A `null` field occupies memory. Yes, it does.

The filter on the `null` type can be also explicitly specified like a query based on the value, because a field of type `null` can have only one possible value which is `null`.

```
db.inventory.find( { item : null })
db.inventory.find( { item { $type : 10 }})
```

Both queries return only the documents with `item = null`. Returning to the collection above:

- The `_id = 1` will be shown.
- The `_id = 2` will not be shown.
- The `_id = 3` will not be shown.

`$exists`

```
db.inventory.find( { item : { $exists : true } } )
```

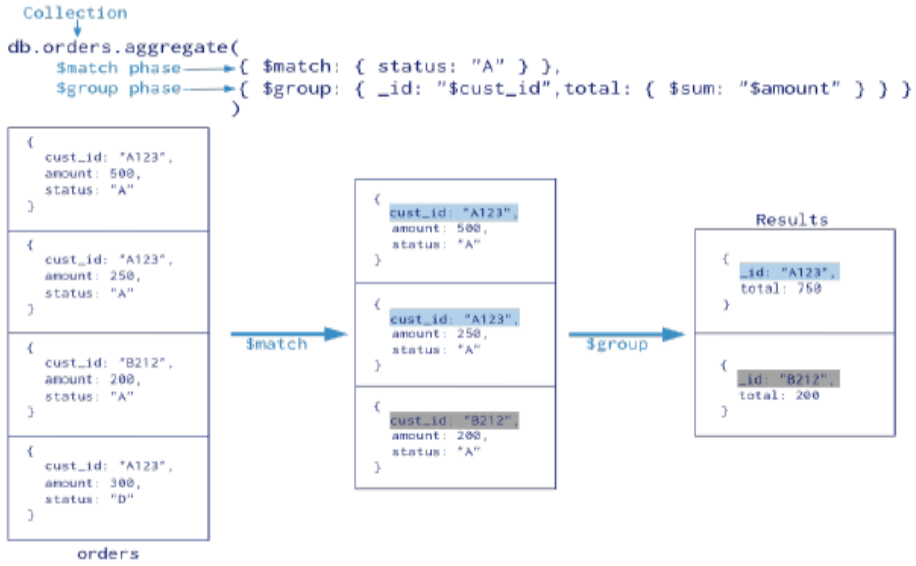
It returns the document only if the `item` field is defined (*it must exist*). Returning to the collection above:

- The `_id = 1` will be shown.
  - Even if it's `null`, the field exists, and it occupies memory.
- The `_id = 2` will not be shown.
- The `_id = 3` will be shown.

Aggregation

Process results sets and return computed results. I can write very complex query by simply creating multiple steps to obtain the final result.

Pipeline



1. `$match`: documents of the collection `orders` are filtered following the condition `status = "A"`.
2. `$group` groups the documents following a field, in this case `_id`.
   a. `$cust_id` will be the new `_id` field name in the result set.
3. Add a new field called `total`, which contains the sum of the field `amount` of each document that remained after the `$match`.

Obvious note about aggregation

This operation is just a query, so the initial collection does not change, it is like SQL, a query does not change the database.

$lookup

Forbidden.

```
{
  $lookup:
    {
      from: <collection to join>,
      localField: <field from the input documents>,
      foreignField: <field from the documents of the "from"
collection>,
      as: <output array field>
    }
}
```

Performs a **left outer join** to a collection in the same database to filter in documents from the "joined" collection for processing. The $lookup stage adds a new array field to each input document. The new array field contains the matching documents from the "joined" collection. The $lookup stage passes these reshaped documents to the next stage.

$unwind

You can use it, but it must be very well justified. The `$unwind` deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

```
$unwind:
   {
     path: <field path>,
     includeArrayIndex: <string>,
     preserveNullAndEmptyArrays: <boolean>
   }
```

The collection is:

```
db.clothing.insertMany([
 { "_id" : 1, "item" : "Shirt", "sizes": [ "S", "M", "L"] },
 { "_id" : 2, "item" : "Shorts", "sizes" : [ ] },
 { "_id" : 3, "item" : "Hat", "sizes": "M" },
 { "_id" : 4, "item" : "Gloves" },
 { "_id" : 5, "item" : "Scarf", "sizes" : null }
])
```

The operation is:

```
db.clothing.aggregate( [ { $unwind: { path: "$sizes" } } ] )
```

The result of the unwind will be:

```
{ _id: 1, item: 'Shirt', sizes: 'S' },
{ _id: 1, item: 'Shirt', sizes: 'M' },
{ _id: 1, item: 'Shirt', sizes: 'L' },
{ _id: 3, item: 'Hat', sizes: 'M' }
```

Aggregation example

Each document represents a neighbourhood.

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

We want to return the **states** with a population greater than 10 million.
For each of them, show the total population.

`$match + $group`

```
> db.zipcodes.aggregate( [
...    { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
...    { $match: { totalPop: { $gte: 10*1000*1000 } } }
... ] )
{ "_id" : "TX", "totalPop" : 16984601 }
{ "_id" : "FL", "totalPop" : 12686644 }
{ "_id" : "IL", "totalPop" : 11427576 }
```

Other examples

`$group + $group`

```
> db.zipcodes.aggregate( [
...    { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
...    { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
... ] )
{ "_id" : "SD", "avgCityPop" : 1839.6746031746031 }
{ "_id" : "NV", "avgCityPop" : 18209.590909090908 }
{ "_id" : "TN", "avgCityPop" : 9656.350495049504 }
{ "_id" : "AR", "avgCityPop" : 4175.355239786856 }
{ "_id" : "WI", "avgCityPop" : 7323.00748502994 }
```

In the query above:

1. We group the neighbourhoods based on the state and the city; then we compute the total population of the city in that state.
2. Then, the result is grouped by the state, and we compute the average population of the cities in each state.

`$group + $sort + $group`

Here we create 4 new attributes:

- The value of the biggest city.
- The value of the smallest city.
- The value of the biggest population.
- The value of the smallest population.

```
> db.zipcodes.aggregate( [
...    { $group:
...       {
...         _id: { state: "$state", city: "$city" },
...         pop: { $sum: "$pop" }
...       }
...    },
...    { $sort: { pop: 1 } },
...    { $group:
...       {
...         _id : "$_id.state",
...         biggestCity:  { $last: "$_id.city" },
...         biggestPop:   { $last: "$pop" },
...         smallestCity: { $first: "$_id.city" },
...         smallestPop:  { $first: "$pop" }
...       }
...    }])
{ "_id" : "RI", "biggestCity" : "CRANSTON", "biggestPop" : 176404, "smallestCity" : "CLAYVILLE", "smallestPop" : 45 }
{ "_id" : "CT", "biggestCity" : "BRIDGEPORT", "biggestPop" : 141638, "smallestCity" : "EAST KILLINGLY", "smallestPop" : 25 }
{ "_id" : "KS", "biggestCity" : "WICHITA", "biggestPop" : 295115, "smallestCity" : "ARNOLD", "smallestPop" : 0 }
{ "_id" : "NC", "biggestCity" : "CHARLOTTE", "biggestPop" : 465833, "smallestCity" : "GLOUCESTER", "smallestPop" : 0 }
{ "_id" : "PA", "biggestCity" : "PHILADELPHIA", "biggestPop" : 1610956, "smallestCity" : "HAMILTON", "smallestPop" : 0 }
```

In the query above:

1. We group the neighbourhoods based on the state and the city; then we compute the total population of the city in that state.
2. We sort the result by the total population in ascending order.
    - So, the first will be the smallest and the last will be the largest.
3. Then, the result is grouped by the state, and we compute the 4 attributes specified above using the keyword of arrays (`$last` and `$first`).

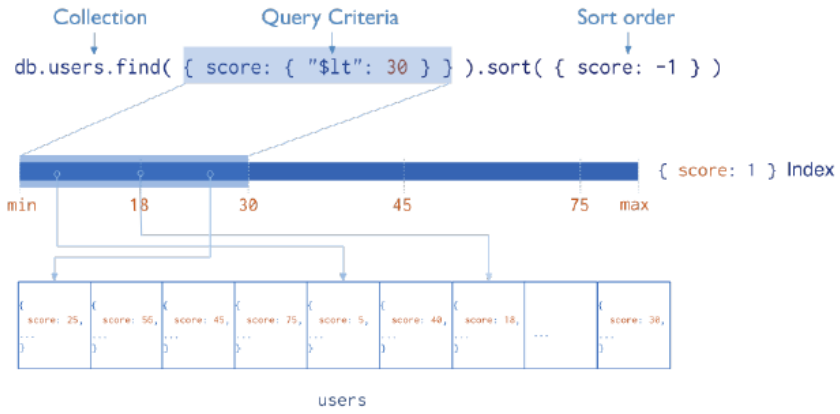
MongoDB CLI commands

`show collections`

Shows the name of all the collections in the database.

MongoDB Indexes

Indexes are a special data structure that store a small portion of the collection's data set in an easy traversal form and are built for speed-up a specific query. Without indexes, MongoDB would have to scan every document in a collection to select the documents that match the condition. The index stores the value of a specific field or set of fields ordered by the value of the field. The index permits also to return ordered result sets.



The index has an entry associated with every value that the field could have. Every entry has a pointer to a document of the collection (*an index works only on the collection level*).

Pros and cons of indexes

Obviously, the index must be maintained. Whenever we get an update or a write operation, we must update the index. An index speed-up a read operation but it can slow-down a write operation, especially if the write operation is frequently performed in the field for which the index has been defined.

The default index

MongoDB has a default index on the `_id` field, which is mandatory for every document. This index prevents clients from inserting two documents with the same `_id`. This index cannot be dropped.

Index commands

`getIndexes`

Shows all defined indexes in the database.

`createIndex`

```
db.users.createIndex( { keys } , { options })
```

The method creates the index only if an index with the same specification (*same* `keys` *and* `options`) does not exist.

```
db.cities.createIndex( { population : 1 } ,
{ name : "query for population" })
```

dropIndex

```
db.cities.dropIndex( "query for population")
```

Can we rename an index?

No, we cannot rename an index once created, we must drop and create it again.

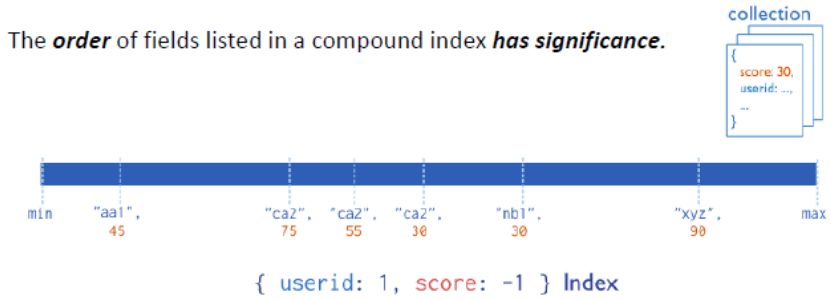
Compound Indexes

What is a compound index?

Index defined on more than one field.

Sorting the keys

```
{ userID : 1 , score : -1}
```
1 → Ascending order
-1 → Descending order.



The order has significance!

Let's suppose that we got an index on the document of the collection `users`, and the index is defined for the keys: `userID` and `score` by using this sort:
`( userID : 1, score : 1)`
When will the index be exploited and when will it be ignored? The index can support a query in which we consider as a parameter (*so in the filtering conditions those two attributes must appear*). Let's suppose we want to get the documents with `score = 1` and `userID = 1`.
`db.users.find( { userID : 1 , score : 1} } )` The index will be exploited.
`db.users.find( { score : 1 , userID : 1 } )` The index will **not** be exploited.

In both cases the result will be the same (*because both queries request the same thing and they are both correct*), but in the second case, the index will not be used. So, the order of parameters in the query counts in compound indexes.

Index for embedded documents

```
{
    "username" : "sid",
    "loc" : {
      "ip" : "1.2.3.4",
      "city" : "Springfield",
      "state" : "NY"
}}
```

`db.cities.createIndex( { "loc.city" : 1 } )` This index will support queries for the attribute `city` of the embedded document `loc`.
`db.cities.createIndex( { loc : 1 } )` In this way, for each document I'll have an entry. This index will support queries on the entire document, but not queries on specific fields of the document.

Index for arrays

```
{
_id : 1 ,
Comments = [
      { test = "aaa" , date : "1022" } ,
      { test = "bbb" , date : "1023" }
]
Ratings = [ 2 , 4 , 9 ] ,
Gros = [
      { test = "A1" , ty = [ 2 , 4 ] } ,
      { test = "A2" , ty = [ 5 , 7 ] }
]
}
```

`db.posts.createIndex( { Comments : 1 } )` In this case the value of the array is a document. The index contains two keys, one for each value of the document.

`db.posts.createIndex( { Ratings : 1 } )` In this case the value of the array is a number. The index contains three keys, one for each value of the array. One key could point to many documents, because more than one document's array could contain a specific value.
`db.posts.createIndex( { "Gros.ty" : 1 } )` In the third case, the index will have one key for every value of `ty`, so one key could point to many documents.

Advanced Indexing

These indexes have not been fully explored during the lectures; they have only been nominated by the teacher. If you want to use them in the project you can do it, but you have to study them on your own.

Geospatial Index

It supports queries based on coordinates.

Text Index

It supports text search (*specific word inside a document*).

Hash index

Build on the hash value of a field.

Types of Indexes

Unique

MongoDB will reject duplicate values for the indexed field. This works also for compound attributes.

Partial

In the index will be inserted only those documents which respect a certain condition. So only a part of the collection will be indexed.

Sparse

The index contains entries only for documents that have the indexed field. So, the index skips documents that do not have the indexed field. Normally, we got a separate entry for null values. A sparse index simply ignores the document.

TTL

These indexes are used to automatically remove documents after a certain period from the collection. This is useful for deleting old logs and session information that only need to persist for a certain amount of time.

In Memory Databases

The choice of the database depends on the application. For the architecture the choice follows the same idea, it depends on the application. In some situations, I do not need to manage a big amount of data (*no big data*), so I can create an application where the entire database can be loaded in the RAM. In some cases, we can distribute the in-memory database in a cluster. Moreover, the memory is becoming cheaper, so it is easy to have a server with RAM of multiple TB.

Features of In-memory databases

Very fast access to the data. Suitable for applications that do not need to write continuously to a persistent medium. Obviously, this kind of architecture is designed to avoid data loss even in the event of power failure.

Solutions for data persistency

Remember that the primary memory is not persistent, wo how to avoid data loss in case of system failure? We got three possible solutions:

Snapshots

A complete copy of the database.

Transaction log

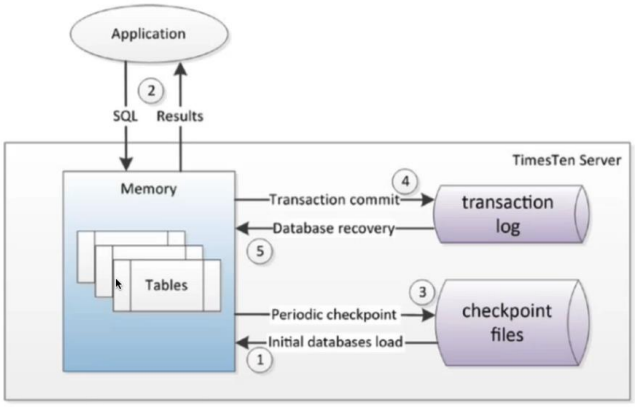
The log of all the transactions computed (*like the commit log*) are stored in a file in the disk. In case of disaster, we can recover the entire data partition.

Replication

Replicate the database in another machine of the cluster.

TimesTen

Produced by Oracles and implements a similar SQL-based relational model. All data is memory resident.



Persistence is achieved in two methods:

Periodic snapshots

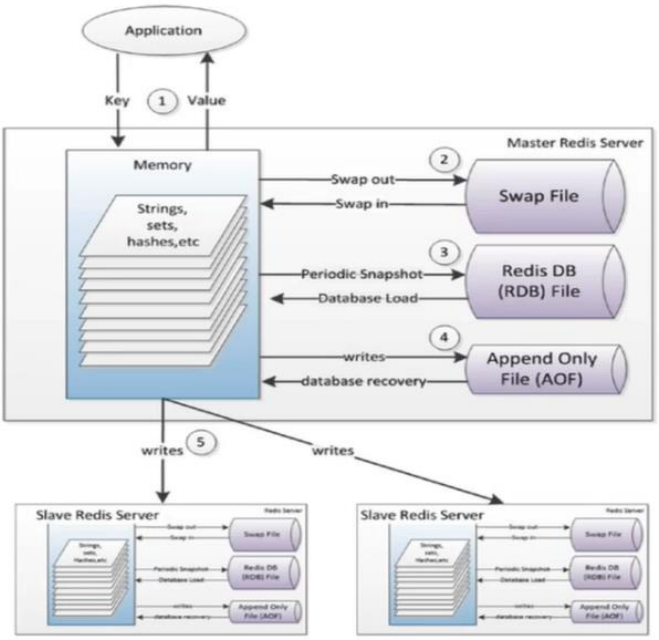
Also called "*Checkpoint files*". Periodically the application does a checkpoint (*a snapshot of the primary memory*) and loads it in the disk.

Transaction log

We write the commits of all the write transactions in a log file (*like the commit log*). The records of the log are like: *"At this time* `t`*, has occurred a transaction with this value* `X`*"*.
By default, the disk writes are asynchronous (*they are performed in a batch way in a second moment*).
To ensure ACID transactions we need to configure the server for synchronous writes to a transaction log after each write, at the cost of a performance loss (*every access to the log is an access to the disk*).

Redis

Redis is a key-value database In-Memory resident. The values that Redis can store consist mainly of strings and various types of collections of strings (*lists, hash maps, …*).



Redis is mostly used for creating queues of requests that can be directed to our real server.

Redis indexes

Only primary key lookups are supported, no secondary indexes are available.

Redis virtual memory (Caching)

It is a mechanism for storing an amount of data larger than the server's primary memory. Old (*less frequently used*) keys are swapped-out to the disk (*in the swap area*) and recovered if needed. At the cost of a loss of performance.

Redis replicas

Redis has replicas to reduce the level of persistence on the master. Obviously, in case of write operation a procedure of eventual consistency must be implemented. If the performance is a very critical requirement and some data loss is acceptable, then a replica can be used as a backup database.

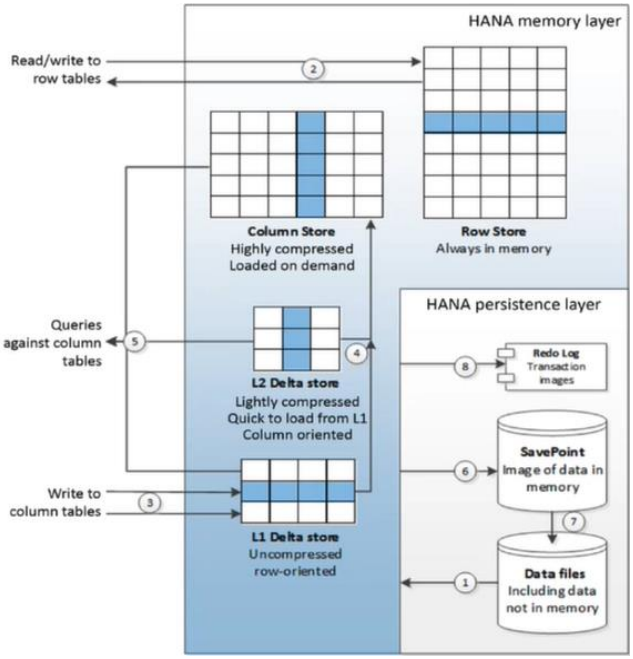
HANA DB

Created by SAP. Is a relational database which combines an in-memory technology with a columnar storage solution. In the case of HANA, we need an optimised hardware configuration.

How does HANA work?

The part stored in columnar organisation is often devoted to analytics. Classical row tables are exploited for OLTP operations (*typical transaction, so write operations*) and are stored in-memory.
However, columnar tables are often used also for OLTP operations.
By default, they are loaded in memory by demand. However, the database may be configured to load in memory a set of columns or entire tables from the beginning.

HANA Architecture



Load the snapshots (1)

The first thing to do is loading the last snapshot from the disk to the memory.

Read/Write operations in the row part (2)

Then, some write and read operations are performed. Those operations are performed in the in-memory part which is organised in rows.

Read/Write operations in the columnar part (3)

The system may be interested in some write operations that write something directly in the columnar part of the database. These operations are recorded in-memory in a cache organised in rows (*no compression*) called `L1 Delta Store`).

Delta stores (4)

To be used for analytics, they are loaded in another cache called `L2 Delta Store`, which is organised in columns instead (*lightly compressed*). Finally, they are loaded in the column store (*highly compressed and better organised*).

Analytics (5)

When we want to make analytics queries, those operations are performed using the Column Store, `L2 Delta Store`, and the `L1 Delta Store`.

Why do we need three databases to perform analytics?

The column store may not be always updated, some information could be in the delta stores.

Persistence (6) (7) (8)

For assure the persistence we got:

- A log of the transaction (*append-only files*), which store all the transaction images (*the value and the timestamp*) (8).
- A subsystem that periodically creates snapshots (6) of the memory (`SavePoint`) and loads them in the disk (7).

Oracle 12c

Where are the files? (1,2)

Data is maintained in disk files but cached in the primary memory.

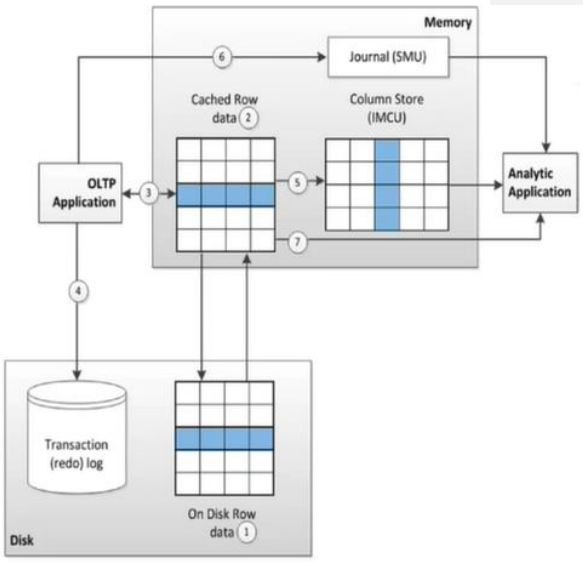
OLTP Operations (3,4)

An OLTP application primarily reads and writes from the primary memory (*in the row part*).
Any committed transaction is always written immediately to the transaction log in the disk.

The updates to the disk row storage are performed in a second moment (by *using the transaction log in case of failure or by simply loading the cache in the memory*).

Analytics (5)

Row data is loaded into a columnar representation (*IMCU – In Memory Column Unit*) for analytics.



Journal (SMU) (6) (7)

Any transaction that are committed once the data is loaded into columnar format are recorded in the journal.

Analytics queries will consult the journal to determine if they need to read updated data from the row store or need to rebuild the columnar store.

Last Update: Thursday, 22 February 2024

Guidelines for selecting a database

Software engineering stuff

Whenever we need to design and develop an application, we need to do some steps, the steps of the software engineering:

1. Requirement elicitation form customers.
2. Requirements definition, both functional and non-functional.
3. Use cases definition, with UML diagram.
4. Identification of the main data structures, main procedures, and relationships.
5. Refinement of the 4° point, create the Database design.
6. Implementation.
7. Test.
8. Deploy.

Technical stuff

Then we need to define some technical stuff:

- Software and Hardware architecture.
- Programming languages and development environments.
- DBMS.

Everything driven by the requirements, common sense, and experience.

Available choices

- Relational PostgreSQL, MySQL
- Key-value Redis, Riak, DynamoDB (*mix of key-value and document*).
- Document MongoDB, CouchDB.
- Column Family Cassandra, HBase.
- Graph Neo4j, Titan.

What guide us?

Non-functional requirements

- Volume of reads and writes
- Tolerance for inconsistent data in replicas
- Availability and disaster recovery requirements
- Latency requirements.

Key-Value databases

Preferrable when:

- The application has to handle frequent but small read and writes operations.
- Simple data models.
- Simple queries.

They do not allow too much flexibility (*on the things we can do, not the flexibility of the scheme*) and the possibility to make queries on different attributes.

Some examples of use

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

Document databases

Preferrable when:

- We got vary attributes, in terms of number and type.
- Elaborate huge amount of data.
- Complex queries (*on different type of attributes*) with the use of indexes and advanced filtering.

Pros

- We got high flexibility on the things we can do.
- High performance and easy to use.
- Embedded documents allow us to store together attributes frequently accessed together.

Some examples of use

- Backend support for sites with high volumes.
- Manage data types with variable attributes, such as products.
- Tracking variable types of metadata.
- Application that uses JSON data structures.
- Application benefiting from denormalization.

Last Update: Thursday, 22 February 2024

Guidelines for the project

Past Questions of the Exam

BigData

1. Main issues? The 5 V's
    a. **Volume**: Many data.
    b. **Value**: Data have a hidden value which must be extracted by algorithm.
    c. **Veracity**: Data could ned to be updated after some time (*so they can have a sort of expire date*).
    d. **Velocity**: Data arrive with very variable and high speed.
    e. **Variety**: Data arrive from different sources and with different format.
2. A solution for manage big data is the scaling:
    a. Vertical Scaling → Upgrade the machines → Can be expensive → Single point of failure.
    b. Horizontal Scaling → Add more machines → Create a cluster of computers → How to manage it? → Distributed Systems.

Distributed Systems

3. MapReduce

Consistency

Strict consistency

When an update occurs, must be forwarded to every replica, the users get notification of the end of the operation only when the update has been made in all replicas.
When a user does a read operation, I am sure that he will read the last version of the data,

Eventual Consistency

The replicas are not updated immediately but they will "eventually" be updated in the future.
When a user does a read operation, I am not sure that he will read the last version of the data,

Causal consistency

A **causal relation** between two operations: if the action A happens before the action B and there is a causal relationship between them, then A must always be executed before B on all replicas, because the relationship itself contains information (*for example: a post and a comment under that post, the comment insertion can't be applied before the post insertion*).
This type of eventual consistency is meant to preserve those relationships.

Read-your-write consistency

When a user does a write operation, I am sure that that user will always read the updated value. Any other client that performs a read operation after the update could get an outdated value, but eventually, they will read the updated value. This consistency is meant to reassure the user that his write operation is successful.

Session consistency

Is like the read-your-write consistency but it regards the session. Any other user in the same session of the first user will read the updated value. Any other user in a different session can

read an outdated value, but eventually, they will read the updated value. If a user changes session it can read outdated values.

Monotonic Write Consistency

If a user makes several update commands, the order of those commands will be maintained even in the replicas.

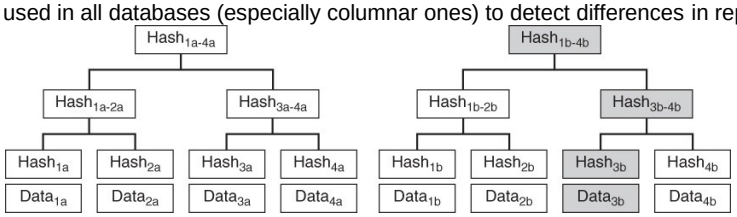
Monotonic Read Consistency

If a user makes several read operations, it will always return the same value or a more updated one (*in the case that another user makes a write operation at the same time*).

Replication

Anti-entropy

Strategy used in all databases (especially columnar ones) to detect differences in replicas.



1. Every server builds its own has tree.
2. Each leaf contains the hash of a partition
3. Each parent contains the hash of the hashes of its children
4. If the root is different, than the two replicas are different, and the oldest one is updated.

Key-Value Databases

Where should you use it?

Document Databases

Where should you use it?

- When I will program using objects (*OOP frustration → in the relational type, entities are divided due to normalisation so entities representation in the code is far than equal respect to the representation in the database*).
- When I prefer a schema-less database instead a relational one (*so I don't need controls on data type neither*).

MongoDB

Columnar Databases

Bloom filter

Last Update: Thursday, 22 February 2024

Graph Databases

InMemory Databases

Other

Software Engineering Recap

"Everyone here takes 30 cum laude at the exam, so I don't need to do this part"

cit. Prof. Pietro Ducange

When you design big applications, you should follow a scheme that ensures a correct production of the software.
The scheme is divided in steps called "*workflows*".
Every workflow requires some time, but some workflows could overlap.

1. **Requirements extraction.**
   - *Often the customer doesn't know what he wants, so be patient.*
   - We must understand what the system must do and who will use it.

2. **Requirements definition.**
   - Define functional requirements and non-functional ones.

3. **Use Case definition.**
   - Expressed with diagrams for a better understanding.

4. **Identification of data structures and main procedures.**

5. **Refinement of data structures and DB design.**

6. **Implementation & Test.**
   - We create/improve a prototype and then we test it.

7. **Deploy the System.**

Functional Requirements

Functional Requirements express what the application should do the job, not how the app does it.
Very high-level states, that express some input-output behaviours, so they describe the output that I desire from the system.
"The system must send a confirmation email whenever an order is placed". (a sort of cause-effect relationship).

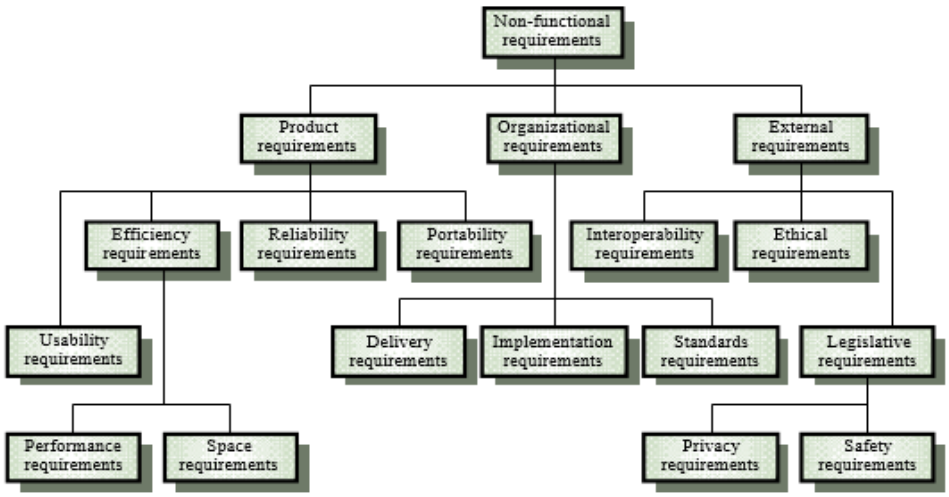
Not Functional Requirements

Not Functional Requirements express properties and constraints, if these are not met then the system is useless.
Can be divided in three main categories:

- **Product Requirements**
  - Requirements which specify that the delivered product must behave way.
    - *"The response time must be below 1 second".*

- **Organisational Requirements**
  - Requirements which are a consequence of organisational policies and procedures e.g., *process standards used*, implementation requirements, etc.
    - "*Must be realized in C++*".

- **External Requirements**
  - Requirements which arise from factors which are external to the system and its development process :
    - "*Legislative*, *privacy* and *safety* requirements".



Use Case and Actors

We define how the system interacts with the external environment.
A use case describes the steps that should be performed to complete a task, in other words how the task should be completed.
We need to identify the actors, prerequisites, post requisites and what is happening in the middle.

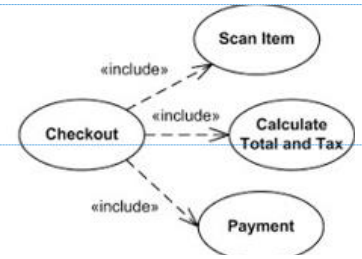
An actor is someone or something that somehow interacts with our system for obtain something.

An actor can really be everything:

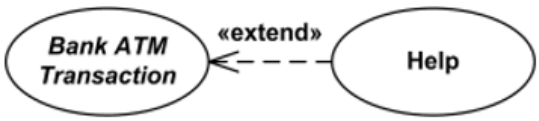
- Human or Animal.
- Peripheral device (hardware).
- External system or subsystem.
- Time or time-based event.

Use case is a sequence of actions that can be expressed with use-case diagram.

Include → A base use case may be dependent on the included use case.
Without the included one the base case is incomplete, because the included use cases represent a subsequence of the base use case.
`Base → Included.`



Extend → The extended use case is dependent on the base use case.
Without the extended one the base case remains complete, because the extended one represents additional functionality.
`Extended → Base`



Commentato [2]: • Use case diagrams describe what a system does from the standpoint of an external observer. The emphasis is on what a system does rather than how.

• Use case diagrams are closely connected to scenarios.

• A scenario is an example of what happens when someone interacts with the system.

Commentato [3]: the inclusion is not exclusive: imagine that we have a base use case like "Checkout" and an included use case like "Payment", so the Payment is related with Checkout but It can be related with another use case like Registration, because maybe the registration implies also a payment of a fee.

Commentato [4]: for example: basic case view item can be extended with add item to the cart , so I'm not obliged to add it for viewing, but it's optional

Data Modelling

Provide a description of the data and the relationship.
Entity sets and relationships sets do not depend on the type of Database and DBMS.
The overall Data Modelling process includes ***three stages***:

Commentato [5]: • Data models provide a structured and formal description of the data and data relationships required for an information system.

• Entity Sets and Relationship Sets do not depend on the type of Data Base and DBMS.

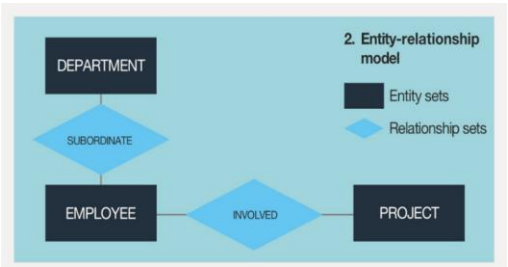
• Data and data relationships will remain stable from the users' perspective throughout the development and expansion of information systems.

Data analysis

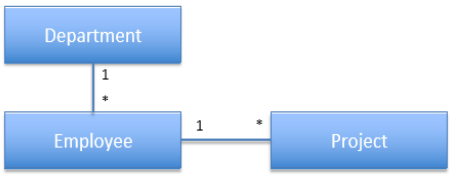
1. Data analysis	Order no. 112 Date: 07/14/2015
Goal	For project monitoring purposes, employees, work, and project times should periodically be logged per department.
1.	Employees report to departments, with each employee being assigned to exactly one department.
2.	Each project is centrally assigned a unique project number.
3.	Employees can work on multiple projects simultaneously; the respective percentages of their time are logged.
4.	...

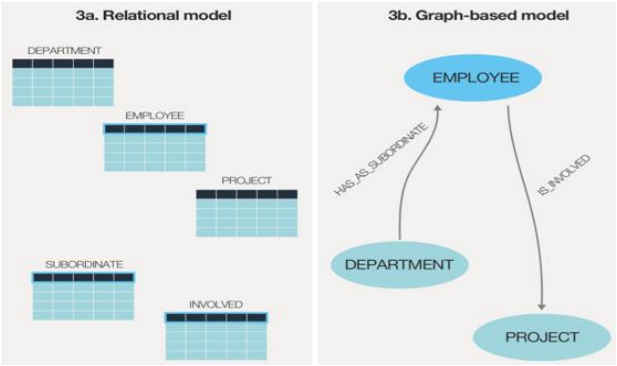
Designing a conceptual data model

Entity-Relationship Model UML Analysis Classes



"*" *equals to "many".*

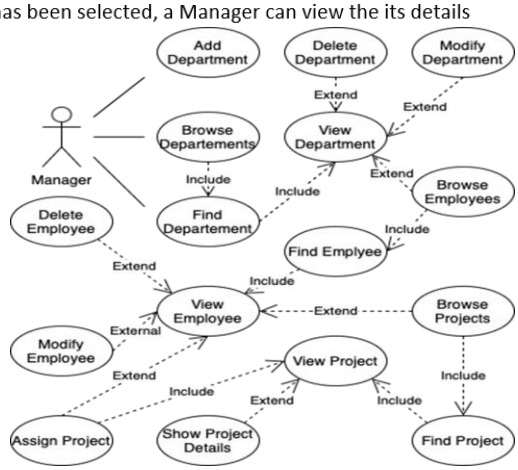




Use Case Diagram Example

Last Update: Thursday, 22 February 2024

1. A Manager can browse the list of Departments
2. A Manager can find a Department (By Name)
3. A Manager can add a new Department
4. A Manager can select and delete a Department
5. A Manager can select and modify the description of a Departments
6. Once a Department has been selected, a Manager can browse the list of its Employees
7. Once an Employee has been selected, a Manager can browse the list of his/her Projects
8. Once an Employee has been selected, a Manager can assign a Project to him/her
9. Once a Project has been selected, a Manager can view the its details



Last Update: Thursday, 22 February 2024

Java Recap

Java is an interpreted language, it is interpreted because it works with a **Java virtual machine** which executes the code, you don't need a specific compiler, it works with different CPUs.

It is strongly object oriented, a robust language, architecture neutral and it is cross-platform.

Some keywords:

- Java is Simple: very high level.
- Java is Secure and Portable: They needed a language capable
- Java is not compiled, but it's Interpreted:
  - By the java virtual machine, which executes the code.
- Java is an Object-Oriented programming language.
  - Java is object oriented, so it's structured in classes.
- Java widely use the concept of class extension.
  - Encapsulation
  - Polymorphism
  - Inheritance
- Java is Architecture neutral and Platform independent.
  - Java is cross-platform because the program runs everywhere there is the Java virtual machine.
- Java has many libraries for distributed programming.

Java can be embedded in web pages; browsers can support Java (it's not used anymore in this way).

A well-organized java program should consist of a set of classes, each class should represent a real entity.
A unique huge class is not good, it is better to have well organized and simple classes.

Every program has a class with a main method, it's the first method that the JVM executes.

In Java there are the concepts of packages and methods:
The **packages** are a way to categorize classes, it is the next level hierarchy after classes.
The **methods** are the functions that enable the interaction of data between classes.

How is a Java file 'compiled' and executed?

Every file contains only one class definition, and the class must have the same name as the '`.java`' file. In Java you should have a file for every class.

The java source code compiler (*javac*) translates the source code to bytes (*java byte code*).
Then the JVM loads and executes the files (*the JVM may compile them to native code internally*).

The main method tells the JVM where to start and execute the program.

This is an example of a class to write Hello world:

```
public static void main(String[] args){
     System.out.println(“Hello world!”);
}
```

If you save this file with the .java extension, if you run in the terminal inside the folder where the .java file is saved, and you write `java file_name.java` the code is executed.

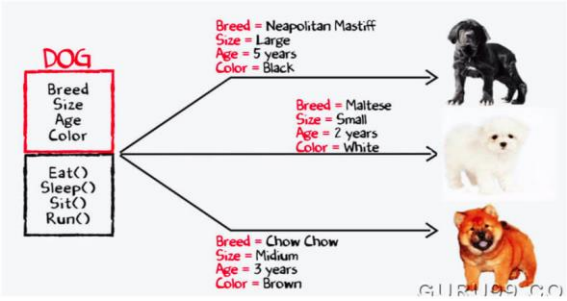
How to Write a Java program

Every class must be defined.
A class has attributes and methods.

```
public class Dog{
      String breed;
      int age;

      public String getBreed(){
            return(breed);
      }
      public static void main(String[] args){
            Dog maltese = new Dog();
            maltese.breed = "Maltese";
            maltese.age = 3;
            System.out.println(maltese.getBreed());
      }
}
```

Encapsulation

Every class has attributes and methods, you create a sort of box which represents data structures you need and methods.
Methods perform operations on the attributes of the instance of the class that called the method.



In red there are data fields and in black the methods, for example you can declare the class below for this example:

```
// Class Declaration
public class Dog {
    // Instance Variables
    String breed;
    String size;
    int age;
    String color;


    // method 1
    public String getInfo() {
        return ("Breed is: "+breed+" Size is:"+size+" Age is:"+age+" color is: "+color);
    }


    public static void main(String[] args) {
        Dog maltese = new Dog();
        maltese.breed="Maltese";
        maltese.size="Small";
        maltese.age=2;
        maltese.color="white";
        System.out.println(maltese.getInfo());
    }
}
```

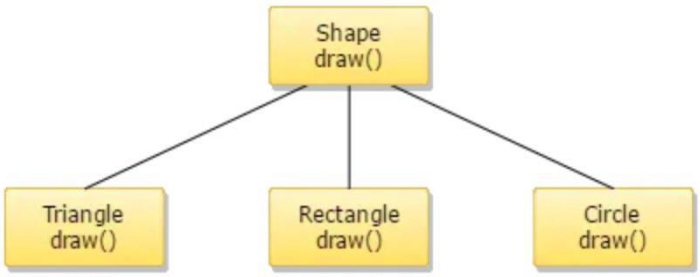
In the main class a new object is instantiated with the **new** operator, and then the attributes of the object are initialized, but this is strange because in every Java program you should use the constructor, you should not allow to directly access the parameters of a class, you must create the constructor, it isn't safe to allow direct access of attributes of a class.

Polymorphism

I can have a variable, function or object that has more than one form.
I can have a common class that specialize in many subclasses, for example:

- Common class: `Shape`
- Subclasses: `Triangle` and `Rectangle`.

I can use the code of `Shape` in `Triangles` and `Rectangles`.



I have the overwriting too; I can redefine methods (so write them differently but using the same name) in the subclasses.

```java
public class Animal {
    public void makeNoise()
    {
        System.out.println("Some sound");
    }
}

class Dog extends Animal{
    public void makeNoise()
    {
        System.out.println("Bark");
    }
}

class Cat extends Animal{
    public void makeNoise()
    {
        System.out.println("Meawoo");
    }
}
```

```java
public class Demo
{
    public static void main(String[] args) {
        Animal a1 = new Cat();
        a1.makeNoise(); //Prints Meowoo

        Animal a2 = new Dog();
        a2.makeNoise(); //Prints Bark
    }
}
```

The superclass defines the characteristics in general, then the dog class is a specific class which is a subclass of the superclass.

We can't assign to a cat an object from an animal class, but we can do the opposite.

The keyword `extends`, extends the properties of a class with all properties of the superclass.

Packages

Java permits you to use a big collection of packages.
I can import classes from another package and use them like they are written by me.

A package consists of a set of classes.

I can use those classes with the command import, which tells the compiler to make available classes and methods of another package, it is the same thing as they include in c++.
A package should contain classes that are related themself.

The main method indicates where to begin executing a class.

References and Primitive data types and Functions

Java has two types of entities.
The primitive type is stored in primitive variables, instead the reference variable stores only the address of the entity.

When i pass a primitive type to a function, the passing is made by value, instead if i pass an object then the passing is made by reference.

Now let's create a new class:

```
public static void modifyNumber(int i){
	int j=10;
	i =i*j;
}

public class Test01{
	public static void main(String [] args){
		List<integer> numbers = new ArrayList<integer>();
		numbers.add(10);
		numbers.add(20);
		System.out.println("Before numbers contains: " + numbers);
		for(Integer i:numbers){
			modifyNumber(n);
}
System.out.println("After numbers contains: " + numbers);
}
}
```

For this method we use the call by value method to pass the variables, only the copy will change, but the array will not be affected by that, so the two println will show the same values.

```
class Person {
	String name;
	String surname;
}
Person(String name){

	this.name=name;
}
public static void modifyName(String newName){

	this.name=newName;
}
public class Test02{
	public static void main(String [] args){
		List<Person> people = new ArrayList<people>();
		people.add(new Person(name: “Alessio”));
		people.add(new Person(name: “Pietro”));
		System.out.println(“Before : ” + people);

for(Person p: people){
			p.modifyName(newName: “NewName”);
}
System.out.println(“After: ” + people);
}
}
```

This time we are passing an object which corresponds to a reference (a pointer in c++), so the output is different between the two println in this case.

The primitive data types are numbers, characters and boolean values.

Integers can be represented by byte, short, int and long;
Real numbers can be represented as float or double;
Characters can be represented as char.

An object is a pointer to the class.

Input in Java

Before the main method we must add the keyword **static** because the main method is the first thing that the Java virtual machine starts executing, without it, the Java virtual machine it would have nothing to start executing from, all methods are called by value.

A method that is not static is an instance method.

```
import java.util.Scanner;
Scanner scanner = new Scanner(System.in);
int numero = scanner.nextInt();
```

Last Update: Thursday, 22 February 2024

Java Code Notes

Try & Catch & Finally block

When executing Java code, different errors can occur coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message.
The technical term for this is: Java will throw an **exception** (*throw an error*).

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.
The `catch` statement allows you to define a block of code to be executed if an error occurs in the try block.
The `try` and `catch` keywords always come in pairs.
The `finally` statement lets you execute code, after `try...catch`, regardless of the result.

```
public class Main {
  public static void main(String[] args) {
    try {
      // Code to try.
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      // Code to execute if an exception occurs.
            System.out.println("Something went wrong.");
    } finally {
      // Code to run in any case.
            System.out.println("The 'try catch' is finished.");
    }
  }
}
```

Throw keyword

The `throw` statement allows you to create a custom error.
The `throw` statement is used together with an **exception type**.
There are many exception types available in Java:

- `ArithmeticException`
- `FileNotFoundException`
- `ArrayIndexOutOfBoundsException`
- `SecurityException`
- …

```
if (age < 18) {
      throw new ArithmeticException("Access denied");
}
```

Connectors

When you need your application to interact with a database, one of the problems that happens is that the types of the database and Java are not one to one, there are some differences, this phenomenon is called Object-Relational Impedance Mismatch.

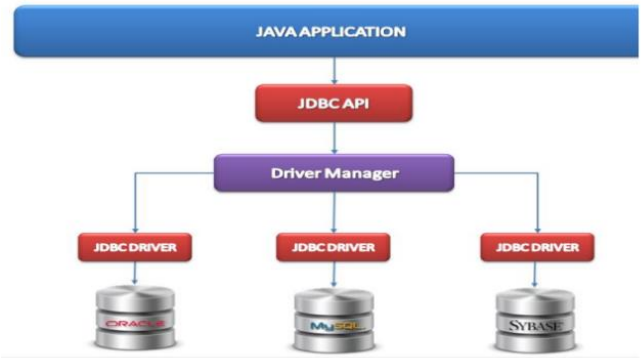
DBMS provide connectors (*drivers*) for certain programming languages.

Each DBMS has its own API (*low portability*), so the application became DBMS-dependent.

Microsoft created ODBC (*Open DataBase Connectivity*) which defines a unified API for accessing databases in C.
A bit later Java introduced JDBC (*Java DataBase Connectivity*), that is equivalent to ODBC, but the APIs are more "java-specific".
The class that contains all of this in the package `java.sql`.

So JDBC is a new layer between my java application and the Driver Manager.

For each DBMS there is a JDBC driver, and each driver communicate with the Driver Manager who it's the only one who can communicate with the application.

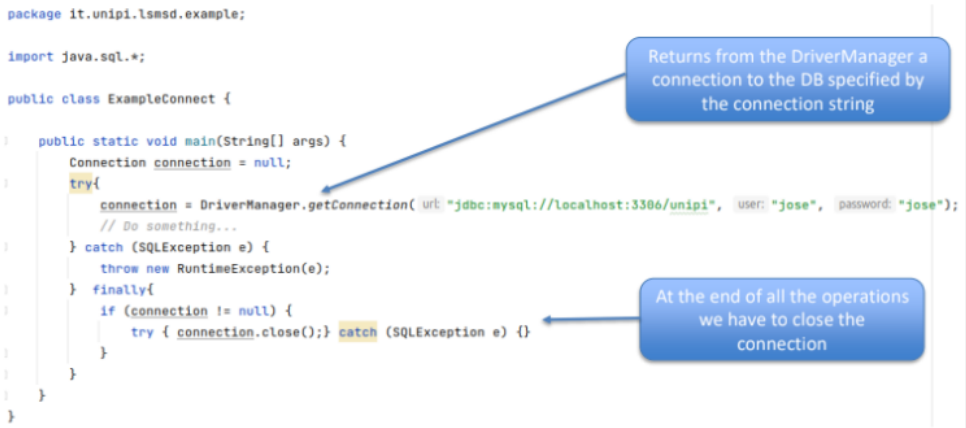


Driver Manager

Responsible of managing all the JDBC Drivers and is the responsible of the connection to the database, it is the main class
The application can set some parameters like (*some of them are obviously mandatory*):

- Type of Driver.
- Type of DBMS.
- IP address of the host where the DBMS is located.
- Credential of a user to connect to the DBMS.
- Name of the Database scheme.

Connection to MySQL



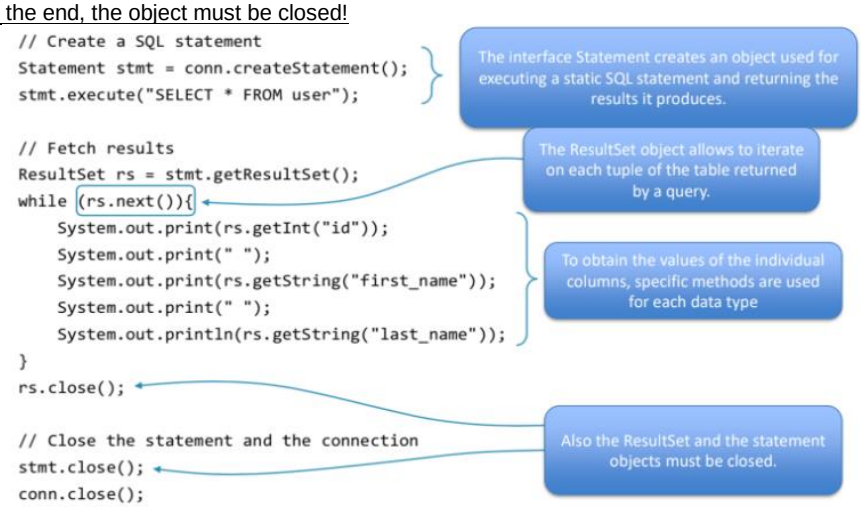
The `getConnection` method creates a connection with the DB, the first parameter is the url of the database, other parameters are the username and the password that must be the credentials of an existing user of the database.

The connection object always needs to be closed at the end of communications.
We put the connection in a `try` for not block the app in case of fail.

Perform Queries and Prepare Statements

After connecting to the DataBase, I have the connection object and for perform queries or operation to the database I must call its method.

At the end, the object must be closed!



I can pre-fill a statement for execute a query multiple time but changing some parameters.

Once you get the connection with the DB you can create interactions with your database, the `createStatement()` function creates a statement object which is needed to perform queries on DB, the resources must be closed in reverse order of the order when they were instantiated.

Update Tables

INSERT, ***UPDATE*** and ***DELETE*** statements do not return a ResultSet object.

The ***executeUpdate()*** method executes the statement and returns the number of rows actually inserted/modified/deleted.

```
String sql = "INSERT INTO user SET username = ?, first_name = ?, last_name = ?, "
                  + "email = ?, password = SHA1(?)";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "alice.ferrari");
pstmt.setString(2, "Alice");
pstmt.setString(3, "Ferrari");
pstmt.setString(4, "alice.ferrari@example.com");
pstmt.setString(5, "1234");

int insertedRows = pstmt.executeUpdate();
System.out.println("Rows inserted: " + insertedRows);
```

Executes the statement and returns the number of inserted rows

? is the placeholder where the parameters are placed, so ? will be replaced.
The `executeUpdate()` method executes the statement and returns the number of rows inserted, modified, or deleted.

If we don't know what kind of operation we make (*insert/update/delete*), we can use the `execute()` method which returns true if it is an update and false if it is a deletion.

Maven

Maven is a software project management tools, which helps you to manage the software production cycle.
Is also a way to standardize these thighs, program, documentation etc.
The main feature of Maven is that you don't need to include libraries manually in your application.
We will use Maven for managing the dependencies in our Java programs (*avoid to manually add/update libraries*).

The POM File

Every Maven project has a POM file.
Maven can be configured through the configuration file, an XML file called "POM file".
This file contains all the configuration details used by Maven for build the project.

When we have a new dependency, we manually add new fields to the dependencies list.

`groupid` □ Uniquely identifies a project across all projects.

`artifactid` □ The name of the jar without the version.

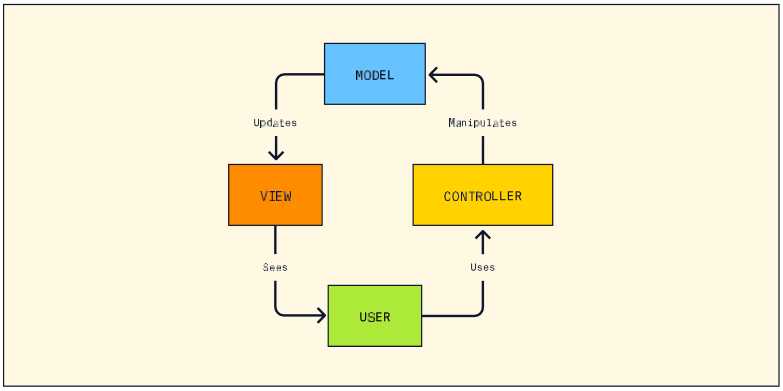
By default, Maven will download form the central repository.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>exercises</groupId>
  <artifactId>exercises</artifactId>
  <version>0.0.1-SNAPSHOT</version>
   <dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
      <version>8.0.17</version>
</dependency>
 </dependencies>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Java MVC

MVC stands for "*Model-View-Controller*" which are the three main components of design analysis of your software, it is commonly used in modern applications.

The idea behind MVC consists in divide the code in sections and each section has a specific purpose.



Model

Encapsulates the data structures of the application.
It is the part of the code that should be responsible to implement the entities.
You should have a java class for each entity, and this should map how you store data in the DB.
The classes in the directory `model` should have:

- Attributes
- Constructor & Destructor
- Get & Set

View

It is the front-end part of the application, so it offers the user interface for the user.
So, it is responsible of the visual output of the application.
Ina web application the view part is composed by `html` and `css` files.

Its main role is to receive the input form the user and to display it with the output in the best way.
It is generally independent (*unaware*) of the application's business logic and data manipulation (*so of the model and controller part*).

Controller

A controller is just a class like others.

A controller is responsible for controlling the way that a user interacts with the application. The controller contains the flow control logic for an application:

- Interacts with the model layer to manipulate data.
  - Returns the information to the user by modifying a value in a class of the model part.
- Interact with the view layer to take the input from the user.
- Defines what is a task and how the task is represented.

Its purpose is to handle and process the input provided by the user (*preprocess input*) and act as intermediary between the user and the models of the applications.

Example of MVC: To-do list application

Requirements

1. This app will let users create tasks and organize them into lists.
2. The tasks could have large font or be a certain colour.

MVC part

- The model in a todo app might define what a "*task*" is and that a "*list*" is a collection of tasks.
  - So, we got a class for the `task` and a class for the `list of tasks`.
- The view code will define what the todos and lists looks like.
  - The `GUI`.
- The controller could define how a user adds a task or marks another as complete.
  - The Controller connects the view's `add button` to the model part (*so to the data*).
  - Example: When you click "`add task`" the model adds a new task in the task list.

DAO & DTO

Let's use the case of a Bookshop application.

Model

In this class are only specified:

- Attributes
- Definitions of Constructors & Destructors
- Definitions of Getters & Setters.

Why only the definition and not the implementation too?
I don't know, the model defines just the possible operations.

DAO - Data Access Object

Every class of the model (`book.java`) has the relative DAO class (`bookDAO.java`).

The baseDAO

In the project there is a class called `baseDAO.java`, where I define a generic connection to the database.
So, it contains common functionalities among the DAO classes.

What contains a DAO class?

Every DAO class is defined as an extension of a baseDAO class.

```
public class PublisherDAO extends BaseDAO
```

The DAO class carries the implementation of the methods defined in the model and have the logic to interact with the objects.
Every time you interact with objects you must use DAO, that is an intermediary between object and logic.

The DAO class and the Database communication

The DAO class also contains all the operation with the database (that concerns the class of the DAO itself).
So, continuing the example of the publisher, the `publisherDAO.java` will contains all the methods for:

- Insert a new publisher in the database.
- Delete a publisher from the database.
- Get a publisher form the database given the `publisherID` and return it like a publisher object.
- …

A few words about the auto commit

Especially when you deal with a RDBMS, it may happen that you have to temporarily turn off all the controls (*so temporarily leave the database in an inconsistent state*) to make some complex operations.

You have to turn off the auto commit option, in this way you can handle data through atomic transactions.
Indeed, if you turn off this option, you have to do the commit manually at the end of the transaction.
So, you can "commit manually" when you finished all the operations on the objects.

Code tips

```
PreparedStatement stat =
      connection.prepareStatement(insertSQL.toString(),
      Statement.RETURN_GENERATED_KEYS);
```

`RETURN_GENERATED_KEYS` are useful when you insert a new record which the primary keys are auto increment, so you avoid making a query to get the just inserted row's id.

```
StringBuilder insertSQL = new StringBuilder();
insertSQL.append("insert into publisher(name, location) ");
insertSQL.append(" values (?,?)");
```

The query has the `?` because it is a parametrized query.

DTO - Data Transfer Object

The DTO is an object with attributes summarized because the author object can have a lot of attributes.

In general, DTO keeps separated model layer and view layer. Sometimes some information is derived from others, so DTO allows to return these derived and computed information.

The DTO generally contains derived information for speed-up operations.

Redis

Can be used like a key-value database, cache, message broker and streaming engine.
It offers atomic operations like incrementing values or calculate hash values.

The database is an in-memory database, it is a data structure storage, allows users to create data structures in memory, it retrieves speed etc, it can be used as a caching system, Redis provides different data types, provides keys that are strings, it supports atomic operations.
Working with a tool that stores data in memory have an issue, when the memory is flushed the data can be lost, but Redis provides a periodicity in which data are stored into the disk, you can set up a **time to live** for your keys. It also provides good scheming features.

For install Redis you go to its site.
Is available for Linux, but you use it in Windows with a Linux subsystem (*WSL2*).

Keys are unique in Redis, they are used to locate a specific value.
Empty string is also a Key.

Keys in Redis are the unique method to locate your files, so it is suggested to assign meaningful name for your keys, a good practice is to use a namespace to separate logical areas for your application, a good practice is to use for the key, the name of the entity, then use the column, then an identifier, then the information you are saving.

When you want to retrieve some information, you have to know the memory location in which it is memorized, you can use a hash algorithm, but the key must be short otherwise it takes a lot of time to compute the hashing.

Use a namespace to identify application keys:
`Sales-app:invoice:10012:status = “PAID”`

Very long keys are not a good choice, because they add latency.

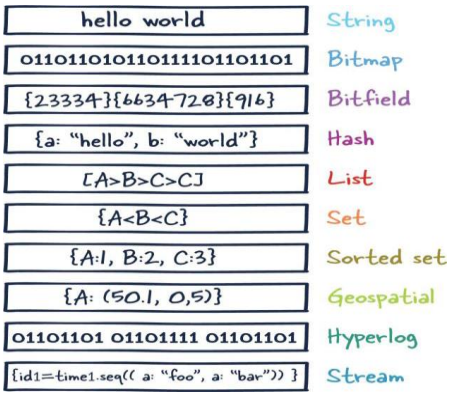
Keys must be readable: “I need a key to store the quantity of followers of a user”
- Good `user:1000:followers`
- Bad `u1000flw`

Dots or dashes are often used for multi-word fields:
- comment:4321:reply.to
- shopping-cart:5431:updated-date

you have a maximum key value size, which is very large.

Redis Data Types

Redis support a certain set of data types:



For our project we can just use strings, but if we act cool, we can use the other too.

Lists and Sets

In list can have duplicates while set can't.

```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> set hello 'hello world from redis'
OK
127.0.0.1:6379> get hello
"hello world from redis"
127.0.0.1:6379> expire hello 10
(integer) 1
127.0.0.1:6379>
127.0.0.1:6379> get hello
(nil)
127.0.0.1:6379>
```

Once you have installed the server, you can connect to the server from another terminal with **redis-cli**.

You can send command to Redis by using a command line interface.

- `Set <key_name> <key_value>` To create a new key.
- `Get <key_name>` To get the value from a key.

The Redis Server

```
saverio@Unum:~$ sudo service redis-server restart
saverio@Unum:~$ redis-cli
127.0.0.1:6379> ping // test connection.
     // results
127.0.0.1:6379> exit
saverio@Unum:~$ sudo service redis-server stop
```

To define a new key-value pair:
`set <key> <value>`

To retrieve the value from a key:
`get <key>`

To define the expiration on a key:
`expire<key> <seconds> NX`
`set <key> <value> EX <seconds>`

To see the time to live of a key:
`ttl<key>`

To delete a key:
`del<key>`

Transactions in Redis

Execution of a group of command in a single step, so in an atomic way.

All the command in a transaction is serialized and executed sequentially in an atomic way (*o tutte o nessuna*).
Using the `EXEC` command, the actions in a transaction are put in a queue and executed one by one (serialized).

Redis do not support rollbacks!! But I can discard a previous sent action using the command `DISCARD`.

Keys involved in a transaction could be modified, so a transaction could influence future transaction.

Let's suppose that we have 2 clients: C1 and C2.

1. C2 send a set of command and execute them first.
2. C2 modified the key `K` in `K'`.
3. After C2 transaction, C1 try to execute its commands.
4. Since `K` has been modified, C1's transaction will be simply aborted.

Round-trip time

Redis is based on client-server architecture and uses TCP connections.

Every command is blocking: any time you issue a command in your client, it will be sent a connection request to the server.
The client program is stopped until a response to the server returns to the client.

The time waited by the client between the instant where the request has been sended and the instant where the server response returns is the round-trip time.

Pipelining

Without pipelining:

```
Client:INCR X
Server:1
Client:INCR X
Server:2
Client:INCR X
Server:3
Client:INCR X
Server:4
```

Pipelining is used to pack all commands together and then receive an answer in a batch.
With pipelining:

```
Client:INCR X
Client:INCR X
Client:INCR X
Client:INCR X
Server:1
Server:2
Server:3
```

`Server:4`
If you have some operations, for example if you have a book and the author and when you create a book, also the author must be inserted, so in Redis all the operations must be atomic, the commands are not executed until you type the `exec` command which will be executed in a sequence in an atomic way.

If you have two terminals that are two clients and on both of them you want to do an atomic operation, and with `watch` command you want to see the same key in these two clients, then you start a transaction in both of the clients, then we execute the command in both of the clients, so both of the commands are queued in both of the clients, then we execute the command `exec` in both of the clients, client 1 done the `exec` command before client 2, then in one of the two terminals we try to retrieve the value of the key the operation in this case will be aborted.

Commentato [6]: Non si capisce

When should you use Redis?
When the application must deal with many volatile data.

Redis is used by many bug apps like GitHub, snapchat etc…

Let's see some scenario: Shop Cart, Online Auction.

Commentato [7]: Mancanti

Jedis

Jedis is a Java library for connecting to Redis and for use it.
All commands are implemented in the class `Jedis`.

```
Import redis.client.jedis.jedis;
String redisHost = "localhost";
Integer redisPort = 6379; // the protocol port of Redis.
Jedis jed = new Jedis( redisHost , redisPort);
// Insert your stuff here.
Jed.close(); // Always close the Jedis object.
```

We must add this dependency to use redis in our Java application.

```
<dependency>
<groupid>redis.clients</groupId>
<artifactid>jedis</artifactId>
<version>4.3.0</version>
</dependency>
```

Connecting to the Redis server

```
import redis.clients.jedis.Jedis;
String redisHost= "localhost";
Integer redisPort= 6379;
//obtaining a single connection to Redis.
Jedis jedis= new Jedis(redisHost, redisPort);
// do your stuff here.
// remember to close all the resources you allocate.
jedis.close();
```

Connecting to Redis server (connection pool)

```
import redis.clients.jedis.jedis;
String redisHost= "localhost";
Integer redisPort= 6379;
//using a pool improves the performance of Redis operations
try (JedisPoolpool = new JedisPool(redisHost, redisPort);
Jedis jedis= pool.getResource())
// do your stuff here.
```

Once you get an instance of a Jedis object, you can execute many commands

```
String key = "my-app:some-key";
//SET command.
jedis.set(key, "Hello world!");
//EXPIRE command.
jedis.expire(key, 10);
//GET command.
String value = jedis.get(key);
//TTL command
long ttl= jedis.ttl(key);
```

We can send multiple commands within a transaction and abort it in case watched keys have been modified.

```
String key1 = "my-app:some-key";
String key2 = "my-app:key-2";
```

Jedis Commands

Watch

If this key is modified, the transaction is aborted.

```
jedis.watch(key1);
```

Multi

```
Transaction transaction= jedis.multi();
transaction.set(key1, "New value for key1");
transaction.set(key2, "New value for key2");
```

Exec

It will return a list with output values of each command sent in the transaction.
The execution of the transaction returns null when keys marked with `WATCH` have been modified

```
List<Object> result = transaction.exec();
```

Connection to the database

[IMG]

Subscribe / Unsubscribe

[IMG]

Publisher / Subscriber

[IMG]

Scaling in Redis

Eviction policies

A configured eviction policy is activated when the max memory limit is reached.
There are different policies:

- **NoEviction**
  - New values aren't saved in memory.
  - If there is replication, this applies to the primary database.
- **AllKeys-LRU**
  - Keeps most recently used keys, removes least recently used keys.
- **AllKeys-LFU**
  - Keeps frequently used keys, removes less frequently used keys.
- **Volatile-LRU**
  - Remove least recently used keys with expire field (TTL) set to true.
- **Volatile-LFU**
  - Removes least frequently used keys with expired field set to true.
- **AllKeys-Random**
  - Extract a random key to delete.
- **Volatile-Random**
  - Extract at random a key with expired field set true.
- **Volatile-TTL**
  - Removes keys with expire field set to true and the shortest remaining time to live (TTL) value.

How to see/set the eviction policy

```
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "0" // No limit.
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "noeviction" // No eviction policy used.
```

Persistence

Snapshots (RDB)

I periodically perform snapshots of my database.
I can lose some write operations that are occurred during the creation of the snapshot.

Append-Only File (AOF)

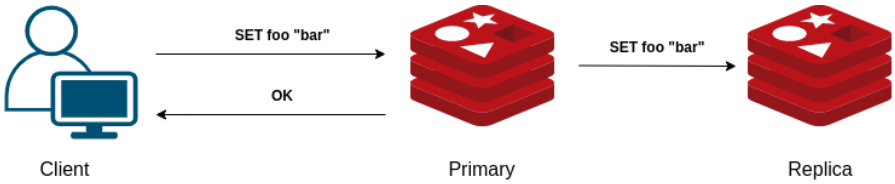
A simple log file, where every write operation is reported.
Using this log, I could rebuild the entire database.

Hybrid (RDB + AOF)

The most expensive (*for performance*) solution, but it's the best in terms of persistency.
I am sure that I won't lose anything anytime.
Because operations that I lose during the snapshot will be saved in the AOF.

Replication

Used to assure availability.
By default, redis works in a master-slave system.
The primary database (master) takes care of keep updated the replicas by forwarding the same write operation to them.
The primary receives the client request, and after some time, will be sent to the replicas too.
When the primary server falls, one of the replicas takes its place.

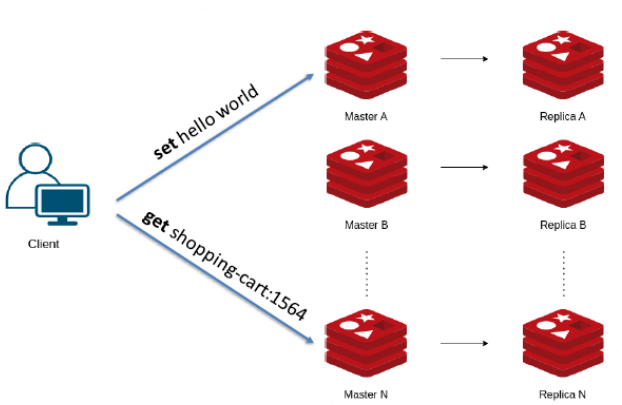


The minimum number of replicas depends on the application.

Clustered database

It is the Redis solution for implementing the scaling.
Keys are distributed among cluster nodes.

How determine where a certain key must go?
To determine in which node a key is stored must be computed a `HASH_SLOT`.
`HASH_SLOT = |CRC16(Key)| 16384 // Modulus.`
There are `16348` hash slots.
Each node of the cluster has assigned a range of slots.



To configure this setup, we must use the `redis.conf` file, in this case we got two nodes, so we assign two ip addresses to them.

We specify that we don't want the protected mode.

Primary, IP: 173.18.15.2	**Replica**, IP: 172.18.15.3
bind **0.0.0.0** protected-mode **no** requirepass **custompassword**	bind **0.0.0.0** replicaof **173.18.15.2 6379** masterauth **custompassword**

Remember: Always set passwords!

Example of replication in redis

We want to create a database with three master nodes and three replica nodes.

```
jose@uss-defiant:~/repository/UNIPI/LSMSD/redis-cluster$ pwd
/home/jose/repository/UNIPI/LSMSD/redis-cluster
jose@uss-defiant:~/repository/UNIPI/LSMSD/redis-cluster$ ls
7000  7001  7002  7003  7004  7005
jose@uss-defiant:~/repository/UNIPI/LSMSD/redis-cluster$
```

Inside each folder (*one folder for each node*) we create the `redis.conf` file with the following content:

```
port 7000
cluster enabled yes
cluster config file nodes.conf
cluster node timeout 5000
append only yes
```

`cluster-node-timeout` is the maximum period in which a node can do not respond, if this time limit is reached, other nodes will consider it fallen.

Start each Redis instance by running the following command in each folder:

```
redis server ./ redis.conf
```

Finally, to create the cluster, run the following command:

```
redis - - cli cluster replicas 1 - - cluster create 127.0.0.1: 7000
127.0.0.1: 7001 127.0.0.1: 7002 127.0.0.1: 7003 127.0.0.1: 7004
127.0.0.1: 7005
```

With the command in bold we specify that we want a replica for every master created.

Test your configuration by creating some keys:

```
jose@uss-defiant:~$ redis-cli -c -p 7000
127.0.0.1:7000> set foo bar
-> Redirected to slot [12182] located at 127.0.0.1:7002
OK
127.0.0.1:7002> set hello world
-> Redirected to slot [866] located at 127.0.0.1:7000
OK
127.0.0.1:7000> get foo
-> Redirected to slot [12182] located at 127.0.0.1:7002
"bar"
127.0.0.1:7002> get hello
-> Redirected to slot [866] located at 127.0.0.1:7000
"world"
```

To stop the cluster run this command:

```
create cluster stop
```

Jedis and Clusters

```
import redis.clients.jedis.*;
import java.util.HashSet;
import java.util.Set;

Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();
jedisClusterNodes.add(new HostAndPort("127.0.0.1", 7001));
jedisClusterNodes.add(new HostAndPort("127.0.0.1", 7002));
jedisClusterNodes.add(new HostAndPort("127.0.0.1", 7003));

try (JedisCluster jedis = new JedisCluster(jedisClusterNodes)){
      /*
       do stuff here
     */
}
```

Notes on Redis Replication

- A master node can theoretically have infinite replicas.
- Redis application is non-blocking on the master side.
- Replication is also largely non-blocking on the replica-side.
- Replication can be used for:
    - *Scalability* Have multiple replicas for read-only queries.
    - For improving data safety and availability.

Keyspace Notification

Redis permits to monitor changes on keys and values in real time.
Clients can receive notification about the expiration of a key by subscribe to the related event channel.

1. When the client creates a new key with a certain TTL.
2. When the key expires, a related event occurs.
3. All the clients subscribed to this event will receive the notification.

This can be useful for some applications.
I can do this thing with every event related to the keys.

Keyspace notification in Jedis

Receiving key space notifications:

```
JedisPubSub jedisPubSub = new JedisPubSub() {
  @Override
  public void onPMessage(String pattern, String channel, String message) {
   System.out.println("Pattern: " + pattern + ", channel: " + channel + ", message: " + message);
  }
};
....
try (Jedis jedis = pool.getResource()) {
     jedis.psubscribe(jedisPubSub , "__keyevent@0__:expired");
}
```

Do not forget to activate the event notification when a key expires. Run this command in Redis:

Publisher – Subscriber

A client can perform the `subscribe` command to a certain channel.
A publication on a channel (performed with the `publish` command) will be notified to every subscribed client.
To stop receive messages, the client can perform the `unsubscribe` command.

To define a subscriber run the following code:

```
JedisPubSub jedisPubSub = new JedisPubSub() {

  @Override
  public void onMessage(String channel, String message) {
    System.out.println("Channel " + channel + " has sent a message : " + message );
  }

  @Override
  public void onSubscribe(String channel, int subscribedChannels) {
    System.out.println("Client is Subscribed to channel : "+ channel);
    System.out.println("Client is Subscribed to "+ subscribedChannels + " no. of channels");
  }

  @Override
  public void onUnsubscribe(String channel, int subscribedChannels) {
    System.out.println("Client is Unsubscribed from channel : "+ channel);
    System.out.println("Client is Subscribed to "+ subscribedChannels + " no. of channels");
  }

};
----
try (Jedis jedis = pool.getResource()) {
    jedis.subscribe(jedisPubSub, "Channel1", "Channel2");
}
```

To publish a message run the following code:

```
try (Jedis jedis = pool.getResource()) {

    /* Publishing message to channel Channel1*/
    jedis.publish("Channel1", "First message to channel Channel1");

    /* Publishing message to channel Channel2*/
    jedis.publish("Channel2", "First message to channel Channel2");

    /* Publishing message to channel Channel1*/
    jedis.publish("Channel1", "Second message to channel Channel1");

    /* Publishing message to channel Channel2*/
    jedis.publish("Channel2", "Second message to channel Channel2");

}
```

Task 3 : Ecommerce

I will have an instance of MySQL (*with replication*) and an instance of Redis (*without replication*).
The application server (*where the java code is executed*) is a single point of failure. The redis server is a SPoF too, because is a unique instance.

So, the new software architecture has:

- Multiple instances of the application server.
- A replica of the Redis server.

The MySQL is essential?
No, we can use other solutions.

Could a columnar database perform better?
According to the requirements, a columnar database can be good.

Software Layered Architecture

Is the architectural pattern of MVC.

- **M**odel
  - It manages the data and the business logic.
- **V**iew
  - It's the frontend part, so, what the user will see.
- **C**ontroller
  - It determines what action a user wants to execute.

MVC is a way to organize the code, for make it more readable.

Which is the path followed by a user's request?

1. **Presentation layer**
   a. ?
2. **Controller layer**
   a. Intercept the request, understand the job to do and then dispatch the request to the proper handler.
3. **Service Logic Layer**
   a. Often coincide with the business logic of the application.
4. **Data Access Layer** (Model Layer)
   a. Layer where entities are implemented.
   b. DAO objects are intermediary between the Service Logic layer and the Data access layer.

Obviously, we can also have more layers.
Like "*Intermediation Layer*", where I put all the API of external components.

Package Organization

The project's code is organized in packages.
Each class is contained in a well-defined package.

Model Package

It contains the implementation of entities involved in the app.
An entity in this package only has attributes, constructors, destructors, and Get & Set methods.

DAO Package

Part of the model layer, where I got DAO objects.

There is one DAO for each model entity.

Base DAO Package

Inside the DAO package there is the Base Package.
It contains the `baseDAO.java` file, so the java class which contains methods that every DAO class have in common.

DTO Package

Part of the model layer, where I got DTO objects.

Some entities could not have a DTO object.

Controller Package

Here is contained the code that handles the input of the user.

Service Layer Package

Here is contained the actual business logic of the app.
If we consider the Ecommerce example: here I can find operation like the "*checkout*".

Redis Exercises

Black Friday

Black Friday, sales increases, CPU utilization in the redis master server was 100%.
How to improve the situation?
We reconfigure the redis architecture to ?? replicas.

Last Update: Thursday, 22 February 2024

Good practices in developing java application

Best practice is a good title, but there are not absolute best practices.
So “good” instead of “best” is a better title.

IA document

Implementation **A**nalysis document is the document that the client will sign with the software house.
The document contains what the software house must develop.

Package structure and naming conventions

How should we organize the hierarchy of directories.

Why should we do all of this?

- **Readability**
  - Well organized code improves the understanding of the code by someone which didn’t write the code.
  - Learning curve of new joiner will be more long.
- **Maintainability**
  - Consume less time for maintain the code.
- **Scalability**
  - Code scalability, which means “*How easy it is to add or change features?*”.

Naming convention of packages

The only best practice about this is the naming convention of packages (and classes).

- Package names is expected to be in lower case (*for avoid conflicts with classes*).
- Companies use their reverse domain name as package prefix:
  - `www.iongroup.com` → `com.iongroup.package_name`
  - If the name contains some java keyword, then we append the underscore “`_`” after the keyword:
    - `example.int` → `int_.example`.
- For internal libraries a similar rule is applied:
  - `[reverse domain][division][lib name]`

How can we organize the hierarchy

There are two or three ways, we will see only two of them.

By Layers (and/or by Type)

Each package contains all the objects that are part of a specific layer of the application.

```
It.unipi.dii.inginf.lsdb.library
.bean
.cache
.persistence
.factory
.model
.service
          .local
          .remote
…
```

By Feature

Each package contains all the object required to implement a specific feature.

```
It.unipi.dii.inginf.lsdb.library
.admin
.login
.registration
.user
.reserve
.search
…
```

Package by Features

Why should we care about hierarchy?

- High modularity, minimal coupling
  - *A good test: what if an entire feature must be deleted, other features will be influenced by this?*
- Easy to read and understand how features are implemented
  - If all the code of a feature is in only one package, then retrieve the code is easier to get.
  - Relationships among features are easier to identify.
    - *For example: the relation between the user and the login functionalities.*
- Package scope should be exploited
  - *If a given object is used only within a package, why should it be visible outside it?*

Defining a hierarchy does NOT mean a given package cannot use items defined in other packages.

We are simply reducing the scope of objects, to prevent abuses.
But if other packages require an object of another package, then we will increase the scope of that object.

Extract API if necessary.

Code duplication and the rule of three

There are programmers that prefer to duplicate the code rather than inheritance.
If we need to duplicate a not complex class, then do it.
But if, then don't do it.

Rule of three: you are allowed to copy and paste the code once, but that when the same code is replicated three times, it should be extracted into a new procedure.

Code duplication is useful when we want to avoid programming many subcases, so avoid writing methods with many parameters.
Only because I don't want to duplicate the code and write only one code for all.

Where place a new piece of code?

Choose the correct package is not an easy task.

- Pay attention to the size of a package.
- Pay attention to package circular dependencies.
  - *Feature in a package A depends on a feature in package B which depends on a feature in package C which depends on another feature in package A.*
  - This thing makes the code harder to debug.

Package by layers

Pros of package by layers

The more natural way to organize applications.

- Item doing the same job are in the same package
  - "*All the caches are in the same package, so when I am looking for a cache, I go in the cache package*".
- Easy to change technology for a single layer.
  - Old and new can even coexist (*in different packages*) while the refactoring is in process.
- Easy to build reusable framework/libraries.
  - Identify the commonalities among different items of the same layer/type and extract an abstract framework for that.

Cons of package by layers

- Hard to understand the implementation of a given feature
  - This approach tends programmers to reuse generic and complex code.
  - Changes can easily break other use cases and the impact is hard to identify.
- Low modularity and high package coupling.
  - Package coupling is here by design, if I structure by layer for sure I will have to include several packages.
- Editing a feature will require changes among several packages and could break other features.
- Difficult to reduce the scope of objects that are supposed to be used un different packages.

Mix-up Techniques

For example, we can organize the first level of hierarchy for features and inside the package of a specific feature I can organize the package for layers.

Don't stick to a solution if you understand that is the wrong choice.

Last Update: Thursday, 22 February 2024

Git and the collaborative work

Pros of Git

- Distributed
  - Each developer has a local copy of the full history.
- Version control
  - History of all changes.
  - Branching
  - Traceability of changes.

How does Git work?

Git works in layers.
Git has many areas that you can call “saving areas”.

Working directory

The actual copy you have access to edit.
So, the file that you see in the file system when you open the folder.

Staging area

Buffer of tour changes ready to commit.
The commit is “*store to remote form the local*”.

```
Git add Myclass.java
Git add .
```

Staging is a concept common in computer science.

Local repository

A full copy of what is available in the remote.
Include all the commits and branches created until the last pull.

So, with the following command I send all the changes in the staging area to the local repository.

```
Git commit -m “A commit message to show.”
```

Remote repository

Centralized copy of your project, hosted in a server.
When we got something committed to the local, after a review all the commits are pushed all together (*in the same order*) in the remote.
Then colleagues can pull the remote copy to their local and see your work.

```
Git push
```

Stash

Parallel buffer you can use to temporary save your changes to the working copy.
Useful when you need to perform operations like merge/rebase from another copy.
Works like a stack, so with pop and push commands.

```
Git stash // push
Git stash pop
```

With a `push` I can put aside your changes to the working copy.
Then with `pop` I can re-apply my changes.

Last Update: Thursday, 22 February 2024

Branch

A parallel code thread that is created from another code thread.
The branch has his own commits.

A branch surely can re-merge with the master copy.
A branch is created when we want to fix a bug or add a feature.

Last Update: Thursday, 22 February 2024

How to structure applications with Maven

How to structure the application and how to split the API and the SPI.

What is Maven?

Maven is a building tool for managing java applications:

- Maven helps you making build process easy.
  -
- Providing uniform build system.
  - It is a big benefit, once you become familiar with the `pom` file syntax and with the most commons plugins that you can use during the maven build phase, you will be able to understand how maven organize a Java project.
- Providing quality project information.
  - I can run programs that evaluate the "*quality*" of my code (*and other statistics*).
  - I can create a set of rules for my code, that the program will check (*a company can create its set of rules that every program written by employees must follow*).
  - Some examples of rules:
    - Max length of a function.
    - Type of parameters.
    - …
- Encouraging better development practices.
  - This point especially refers to the testing part.
  - Maven gives us the opportunity to create a separate directory where to put the code to test.
    - So, we can separate the code used for testing (*under test code*) from the code that is being written by employees (*production code*).

POM file - How Maven works?

Maven is based on the pom (*Project Object Model*) file.
There is a `pom.xml` file containing information about the project and configuration details used by Maven to build the project.

Some POM tags

- modelVersion: the version of the maven model
- groupId: unique Id of the organization/group that created the project
- artifactId: unique base name of the artifact generated by the project
- version: version of the generated artifact
- name: display name of this project
- url: where this project can be found (useful for documentation)
- properties: values whose scope is the current pom
- dependencies: external libraries to be used in the project
- build: contains build technical stuff, including the managing of plugins

Properties tags

You can think about `properties` like variables that are visible inside the `pom.xml` file.
You can use properties in other section of the file like variables in the code.

Properties are useful to avoid hard coding in the pom file!

Dependencies tags

In these tags you can specify external libraries that we want to import in the project.

Build tags

Contains build technical stuff, so:

- Plugins used in the project (*and their configuration*).
- Source template.
- Profile of building (*useful to custom the building process, so to avoid building the entire project every time*).

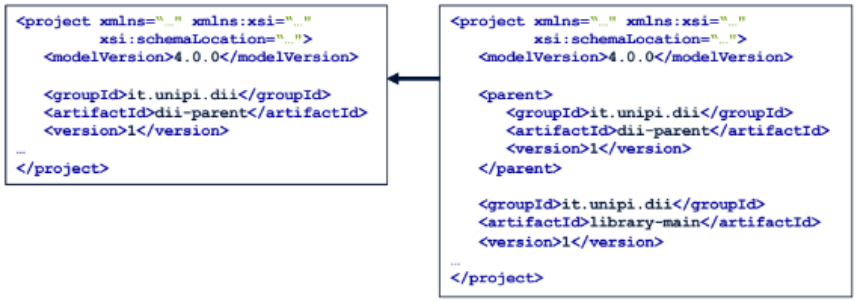
Last Update: Thursday, 22 February 2024

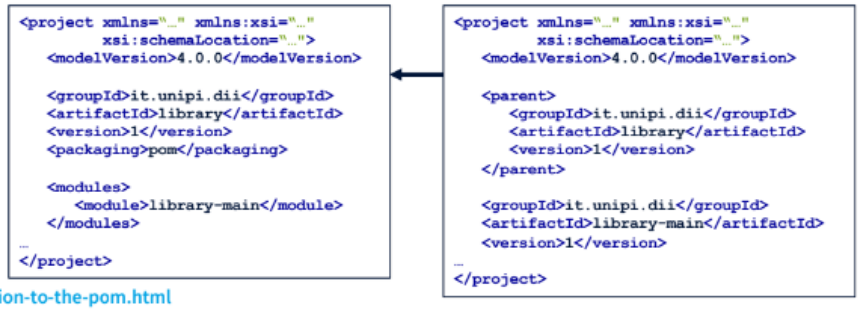
How to create multi-module projects – the parent POM

The parent pom is a normal POM file that we can refer to use it in other applications.
In brief, our POM (also *called "child POM"*) will inherit some tags from the parent POM.

The parent POM is also called "*Super POM*".

You can see like a class (*parent POM*) and its sub-classes (*all the child POM's*).





:ion-to-the-pom.html

If we got a parent POM, we could avoid writing some tags in the child POM(*like the* `project` *tag*).
However, in the child POM I can still overwrite tags that are contained in the parent POM.

How to use a parent POM?

A parent POM is useful for aggregate modules of the project.

```xml
<project xmlns="..." xmlns:xsi="..."
         xsi:schemaLocation="...">
  <modelVersion>4.0.0</modelVersion>

  <groupId>it.unipi.dii</groupId>
  <artifactId>library</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>library-main</module>
    <module>library-common</module>
    <module>library-books</module>
    <module>library-music</module>
  </modules>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
...
</project>
```

it section

```xml
<project xmlns="..." xmlns:xsi="..."
         xsi:schemaLocation="...">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>it.unipi.dii</groupId>
    <artifactId>library</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <groupId>it.unipi.dii</groupId>
  <artifactId>library-main</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>it.unipi.dii</groupId>
      <artifactId>library-common</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>it.unipi.dii</groupId>
      <artifactId>library-music</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit-jupiter-api</artifactId>
    </dependency>
  </dependencies>
...
</project>
```

Figure 9 At the left we got the parent POM; at the right we got the child POM.

Why should we do this?

The code base, during the production, can significantly grow.
The longer the application lasts, the huger is the amount of code, so the more complex will be the maintenance.

How can we link the child POM and the parent POM?

In the example on the right (*the child POM*) we can see the tags `<parent></parent>` , with these tags I can specify which is the parent POM of this child POM.

You can also specify the path of the parent POM.
`<relativePath>../baseapp/pom.xml</relativePath>`

In the example on the left (*the parent POM*) we can see the tags `<packaging>pom</packaging>`.
With these tags I can specify that the project have multiple modules.

How can I include modules?

In the example on the left (*the parent Pom*) we can see a sequence of `<module></module>` tags.

With these tags I can specify the modules that I want to be included.
Mind that every submodule represents a specific concern of the application.

Then in every child pom I can specify (*with* `<dependencies>`) the modules that I want to use in the project (*related to the child POM*), like a library.

Can I avoid specifying common dependencies in the child POM's?

Obviously yes!
If in the parent POM I specify some dependencies (*inside the tag* `<dependencyManagement></dependencyManagement>`), every child POM linked to the parent POM have only to specify the `<groupId>` and the `<artifactId>` in the tag `<dependency>`.
Other information about the dependencies will be taken from the parent pom.

Version control of dependencies

Let's suppose that you ran a java application related to a child POM.
And let's suppose that you have specified some modules in the parent POM.
Obviously, the child POM will inherit the tags.

The java compiler will try to find the module.
If in the child POM you don't specify a version: the first model that it finds, it is the module that will be loaded.
For this reason, you must take care about older versions of the libraries.

But if you specify the `<version>1.0-SNAPSHOT</version>` in the child POM, then it will be loaded the `<version></version>` specified in the parent POM in the dependency's tags.

How to build a framework

A framework is a shared set of libraries (*like code for memory cache or math functions*).

In the parent POM (*the image on the left*):
`ing-inf-parent` is the framework name, and stuff in the `<module>` tags are the libraries included in the framework.
Each module is a library.

Then in the child POM (*the image on the right*):
We specify with `<parent>` the POM file, with the version of the pom file.

In the child POM we specify the libraries (*offered by the framework*) that I want to use.

In this way, we are sure that we always use the latest version of the libraries because in the child POM we have specified that we use the version specified in the parent POM.

Last Update: Thursday, 22 February 2024

API & SPI

API (Application Programming Interface)

Why should we care about this?

The API of a library/module is the set of publicly accessible set of items provided by an external system.

API, give you access to functionalities required by an application.
Moreover, API separate public items (*interface, shared beans*) from implementation details.

In poor words, you are forcing the programmer to use the API and not to use the entire implementation (*which remain hid*).

In the code, you specify the definition of the objects that we want to use.
The actual implementation is in another directory.

In API addition is not a problem, removal is.

SPI (Service Provider Interface)

Publicly accessible set of items that can be implemented or extended to achieve goals.

The application is responsible for the implementation of a specific interface invoked by the library.
This is useful, because some part of the library must be application-specific, but we still want to use a library (*which is a set of general-use code*).
The application-specific part is the SPI.

In SPI removal is not a problem, addition is.

Code Obfuscation

It is always possible to decompile the java bytecode and get the source code.
To avoid this, companies often adopt java code obfuscation.

Code obfuscation consists of:

- Files, classes, methods, and fields are renamed with random identifiers.
- Comments removed.

We can use some tool for this, like ProGuard.

The problem about this technique is that we can have problems with libraries and multi-module maven applications.
Because code obfuscation also changes the name of the function of the libraries.

How can we solve this problem?

We can simply specify which modules don't need to be obfuscated.
I can do this thing with the keywork `@KeepName`.

```
// File: PriceGridServiceImpl.java

package it.unipi.dii.inginf.lsdb.library.grid;

import …

class PriceGridServiceImpl implements GridService {

    PriceGridServiceImpl() {}

    public List<String> getColumnNames() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```



```
// File: b.java

package it.unipi.dii.inginf.lsdb.e.a;

import …

class b implements c {

    public b() {}

    List<String> d() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

```
// File: PriceGridServiceImpl.java

package it.unipi.dii.inginf.lsdb.library.grid;

import …

class PriceGridServiceImpl implements GridService {

    PriceGridServiceImpl() {}

    public List<String> getColumnNames() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

```
// File: b.java

package it.unipi.dii.inginf.lsdb.e.a;

import …

class b implements GridService {

    public b() {}

    List<String> getColumnNames() {
        return Lists.newArrayList("ItemId","Price");
    }
}
```

```
package it.unipi.dii.inginf.lsdb.e.a;
import …;

@KeepName
@KeepPublicClassMemberNames
public interface GridService {
    List<String> getColumnNames();
}
```

Javadoc – Document your stuff!

It is a good practice to document public API using Javadoc.

When you are using IntelliJ and you go over a function with the pointer of the mouse, you can see a little description that explains parameters and other stuff; that thing is very useful, and it is made with Javadoc.

A documentation of Javadoc is made with HTML tags.

```
/**
 * Manager for Multimedia objects.
 * @since  2.4.1
 * @see   {@link Multimedia}
 */
public class MultimediaManager {
   //… private part

   /**
    * Processes a new Multimedia and its price.
    * <p>
    * Depending on the price value:
    * <ul>
    * <li>p>0: we do this and not that</li>
    * <li>p=0: we do that and not this</li>
    * <li>p<0: we do both this and that</li>
    * </ul>

    * @param m  the Multimedia to be processed, not null
    * @param p  the Multimedia price
    * @throws {@link MultimediaException} if m is null or invalid
    */
   public void process(Multimedia m, double p) throws MultimediaException {
      //…
   }

   /**
    * Reads a Multimedia provided its unique identifier

    * @param id  the unique identifier of the Multimedia
    * @return the Multimedia corresponding to the id, null if not found
    */
   public Multimedia getMultimedia(String id) {
      //…
   }
}
```

Javadoc has some interesting tags like:

- `@param`
- `@return`
- `@throws`
- `@link`
- `@code`
- `@see`
- `@since`
- …

Testing Java applications

Smoke testing

"Smoke testing" → Manually test the application after a few changes.

Smoke testing exists and we can use it (*the classic* `printf` *test*), but it is not enough!

Unit test

What is a Unit test?

A unit test is a procedure to verify that a particular detail of a module is working properly.

A unit test does not to test the whole application!

It is written by developers (*often the same developers which wrote the code to test*).

Unit tests must be documented (*like normal code*) and written along with the code (*tests must be done during the code production and not at the end*).

Moreover, when you add a new functionality, it's a good practice to test the old functionalities too (*for be sure that aren't conflicts*).

A test is not a unit test if:

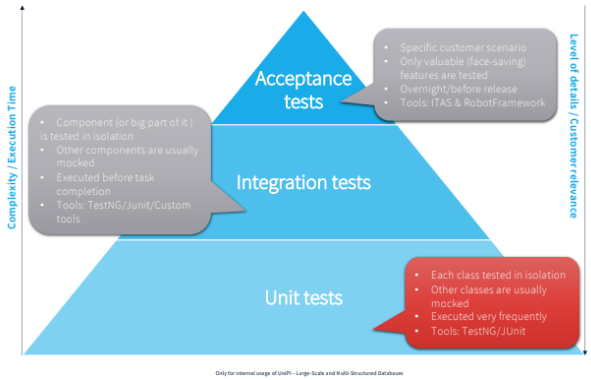
- It communicates across the network.
  - We want to test our code (*behaviors*), not the third-party role!
- Touches the file system.
- Somehow talks to the database (*read or write*).
- Runs threads.
- Forces you to do special things to the environment (*such as editing config files*) to run it.

A test is a unit test if:

- It is quick, a unit test does not take ages to be performed.
- It is part of the build in Maven.

Last Update: Thursday, 22 February 2024

Testing Phases



Acceptance test

The most time-consuming ones.
For this reason, it is usually done overnight or before the official release.
Tests a specific customer scenario, so it tests many modules, classes, third-party components, and the interactions between them.
But only valuable features are tested.
Tools: ITAS and RobotFramework.

Integration test

Component (*or big part of it*) is tested in isolation.
Other components are usually mocked.
It is usually executed before a task completion.
Tools: TestNG, JUnit and other custom tools.

Unit test

It is the cheapest one.
Each class (*module*) is tested (*in an automatic way*) in isolation, and other classes are usually mocked.

A unit test is executed very frequently.
Usually, a unit test is very simple, almost "*stupid*" (*because it must be very fast*).
A unit test is useful for immediately discover little or big bugs.
Tools: TestNG and JUnit.

Best practices for unit tests

One class, (at least one fixture)

Add a fixture for each class that you add to the project.
For different scenarios, use different fixture.

One test, one assert.

For the "*single responsibility principle*" every test should contain only one assert.
So, you should write many tests against the same class method.

Each test must be short

Max 10 lines of code for each unit test.

Test name must clearly specify the goal of the test.

No generic names, better a long and verbose name.
`Test1` □ NOT OK!
`GIVEN_author_names_available_WHEN_search_by_prefix` □ OK!

Test must be easy to maintain

DRY Principle □ "Don't Repeat Yourself".
KISS Principle □ "Keep It Simple and Stupid".

Test must be simple

Avoid complex logic (`if`, `switch`, `for` …).

Avoid dependencies between tests

A test should be able to stand on its own.
Its result must not depend on the result of another test or on the order which tests are performed.

Tests must be fast and automatic

Avoid special configurations.

Test boundary cases for each types of data

Numbers (0, positive or negative), Strings (length, empty…), Dates, Collections (empty, first, last…), exemptions thrown…

Write a test for a bug

If a test make you discover a bug, then you should write a specific test for that bug.

Cover the code properly

Don't leave the under-production code uncovered.

You break it, you fix it

Who breaks a test is also responsible to fix it, as soon as possible.
A test is broken if after a change in the code, a test that before was successful performed now it fails.
This thing is called "*regression*".

Tests must be easily readable

The most important one.

A test is a documentation about a specific feature.

Common approaches to testing

Noobs approach

Code for days, test for days.

This approach is expensive and hard.

Better approach

Code a little, test a little.

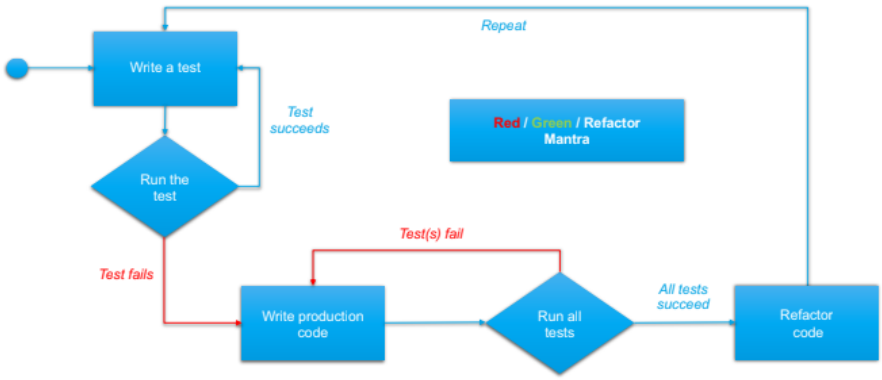
This approach permits to:

- Spot immediately design problems.
- Recognize early unused or useless code.
- Intercept errors as soon as possible.

Optimal approach

You use the TDD (*Test Driven Development*).
Write a test first, and then write the minimum amount of production code to let the test passes; then repeat.

Write the test, write the code that pass that test.



In this way, I will always have total coverage of all the production code!

Remember: No new code shall be written without associated test.

Last Update: Thursday, 22 February 2024

JUnit

It is an open-source framework for perform testing in java applications.

Provides annotation to identify test methods:

- `@Test(`*this function is a test*`).`
- `@BeforeEach (`*executed before every test*`).`
- `@BeforeAll.`
- `@AfterEach (`*executed after every test*`).`
- `@AfterAll.`

Junit provides assertion for checking expected results.
`Assertion.assertEquals( a , b );`
The assertion (and so the test) fails if a and b are different.

A test with Junit can be run automatically and provides immediate feedback after checking the expectations.

How to import JUnit:

Parent POM

```
<properties>
    <junit.platform.version>1.6.2</junit.platform.version>
    <junit.version>5.6.2</junit.version>
</properties>
<dependencyManagement>
    <dependencies>
        … other dependencies not scoped for tests, e.g. guava
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-commons</artifactId>
            <version>${junit.platform.version}</version>
            <scope>test</scope>
        </dependency>
...
```

Module POM

```
<dependencies>
    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
    </dependency>
</dependencies>
```

```
...
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-engine</artifactId>
            <version>${junit.platform.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

`@VisibleForTesting` → Annotation for increase the scope of the function.

Mocking

A mock is a substitute implementation of a class to emulate the code.
A mock must be used when a class is:

- Expensive or impractical to instantiate.
- Belongs to a third-party library.

A mock object must be:

- Simple.
- Shouldn't contain any kind of logic.
- Allow to setup private state.
- Be stupid!

There are some frameworks for mocking (*or I can do this manually*).

```
public interface IEngine {
   void start();
}

public class EngineImpl implements IEngine {
   public void start() {
      Engine.getInstance().start();
   }
}

public class MockEngine implements IEngine {
   boolean started = false;

   public void start() {
      started = true;
   }

   public boolean wasStarted() {
      return started;
   }
}
```

Mocking Frameworks (Mockito)

Sometimes manual mocking is not possible.

Mockito is a framework that allows us to automatically write mocks that emulates interfaces or classes.

Parent POM

```
<properties>
    <junit.platform.version>1.6.2</junit.platform.version>
    <junit.version>5.6.2</junit.version>
    <mockito.version>3.0.0</mockito.version>
</properties>

<dependencyManagement>
    <dependencies>
        … other dependencies not scoped for tests, e.g. guava
        … other dependencies scoped for tests, e.g. junit
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>${mockito.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-junit-jupiter</artifactId>
            <version>${mockito.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Module POM

```
<dependencies>
    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
    </dependency>
</dependencies>
```

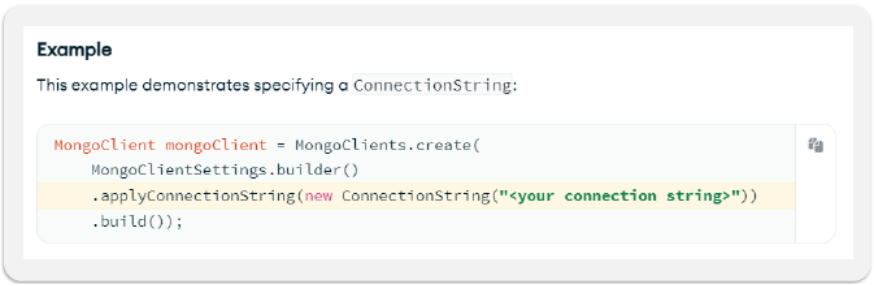
But a mocking framework has some cons:

- Tests become fragile.
- It requires more knowledge of the code under test.
- Lower readability.
- Does not encourage the sharing of mocks in the team (*everyone tends to have their own mocks*).

Java & MongoDB

Install the community edition of MongoDB.

MongoClient



`MongoClient` provides an interface for the MongoDB instance.
The `MongoClient` object represent a pool of collections to the database.
We will only need one instance of class `MongoClient`, even with multiple threads.
The `MongoClient` object is thread-safe, multiple access by different threads to the single instance is managed by the class itself.

MongoClientSettings

To control the behaviour of the `MongoClient` I can use an object called `MongoClientSettings`.
That object is returned by the `create` method.

close()

Method used at the end of the application.
It cleans up resources.

Access to the database

getDatabase()

Method used to access the database.
`MongoDatabase MDB = MongoClient.getDatabase( "testDB" );`
If the database does not exist, it will be created (*confirmed by the MongoDB documentation*).

Access and handling a collection

getCollection()

To access a collection, you must use:

```
MongoCollection<Document> MC = MDB.getCollection( "users" );
```

If the collection does not exist, it will be created.

listCollectionName()

Get the list of existing collections:

```
for(String name : MDB.listCollectionName() ) {
     System.out.println( name );
}
```

drop()

It drops the collection.

```
MC.drop();
```

Create and insert a document

Document class

To create a document, you can use the class `Document`:

```
Document doc = new Document( "name" , "MongoDB" )
      .append( "count" , 1 ).append( "version" , 2 )
      .append( "Name" , "Sergio" ).append( "Surname" , "Arrighi" )
      .append( "State" , "Italy" ).append( "City" , "Pisa" )
      …
      .append( "Friends" , Arrays.list( "Paolo", "Luca" );
```

insertOne()

Then, just insert it into the collection:

```
MC.insertOne(doc);
```

insertMany()

Then for insert it into a collection you will use:

```
List<Document> docs = new ArrayList<Document>();
/* populate docs */
MC.insertMany(docs);
```

Query a collection

`find()`

You can retrieve a single document in a collection (*on a* `MongoCollection` *object*).
You can pass a query filter to the `find()` method to query for and return documents that match the filter in the collection.
If you do not include a filter, MongoDB returns all the documents in the collection (*it means no filters installed*).

```
MC.find( and( gt( "i" , 50 ) , lte( "i"  , 100 ) ) );
```

"Document with 50 < i <= 100"
The ',' means 'and'.

`first()`

The `first()` method returns only the first matching document.

Find and print documents

```
try (MongoCursor<Document> cursor = myColl.find().iterator())
{
   while (cursor.hasNext())
   {
      System.out.println(cursor.next().toJson());
   }
}
```

`Consumer Class`

Used to iterate through query results.

```
import java.util.function.Consumer;

Consumer<Document> printDocuments = doc ->
      {System.out.println(doc.toJson());};

myColl.find().forEach(printDocuments);
```

Collect results in a list

```
List<Document> results =
    myColl.find().into(new ArrayList<>);
```

Update a document

UpdateResult class

```
UpdateResult updaters = Collection.updateFunction(
     <filter condition>,
     <update>
);
```

The update function returns an object `UpdateResult`, which gives us some information about the operation, included the number of documents modified.

updateOne()

```
MC.updateOne(eq( "name" , "Alessio" ), set( "age" , 28 ));
```

updateMany()

```
MC.updateMany(lt( "age" , 28 ), set( "name" , "Alessio"));
```

Delete a document

DeleteResult class

```
DeleteResult deleters = Collection.deleteFunction(
     <filter condition>
);
```

The delete function returns an object `DeleteResult`, which gives us some information about the operation, included the number of documents deleted.

deleteOne()

```
MC.deleteOne(eq( "name" , "Alessio"));
```

findOneAndDelete()

```
MC.findOneAndDelete(eq("name" , "Alessio"));
```

This command deletes the document, but also returns it.

deleteMany()

```
MC.deleteMany(gt("age" , 90));
```

Pipeline

match()

I can pipe together several matches, to combine them and use them in a unique query.

```
Bson match1 = match(eq("categories" , "Bakery"));
Bson match2 = match(lt("year" , 1980));
…
Bson matchN = match(eq( "City" , "Rome" ));
```

group()

```
Bson g1 = group("$city" , sum( "totPop"  , "$pop" ));
```

project()

How to specify the subset of field to return?

```
Bson p1 = project(fields(include( "city" )));
Bson p2 = project(fields(exclude( "_id" )));
…
Bson p1_2 = project(fields(
exclude( "_id" ),include( "city" , "state" )));
```

sort()

```
Bson s1 = sort(descending( "pop" ));
Bson s2 = sort(ascending( "avgAge" ));
```

limit()

```
Bson l1 = limit( 2 );
```

skip()

```
Bson sp1 = skip( 2 );
```

unwind()

It deconstructs the array field and returns several documents having:

- As fields the same fields of the original document.
- Instead of the array field, the document will have a new one which contain one of the values inside the array.

In other words: Let's suppose that the array has 3 elements.
The `unwind` will return 3 different documents, each with one of the 3 elements of the array.

```
Bson uw1 = unwind( "grades" );
```

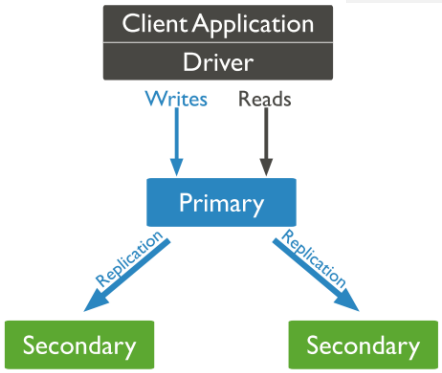
How to use the pipeline?

```
coll.aggregate(Arrays.asList(matchN,g1,p1_2,s1,l1,uw1))
.forEach(printDocuments());
```

MongoDB Replication

The primary node

We can have only one primary node.
The primary node is the node which receive the write and read operations by the clients.
When a write operation occurs, also the replicas will be updated following a certain consistency policy.

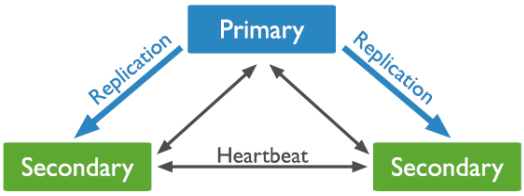


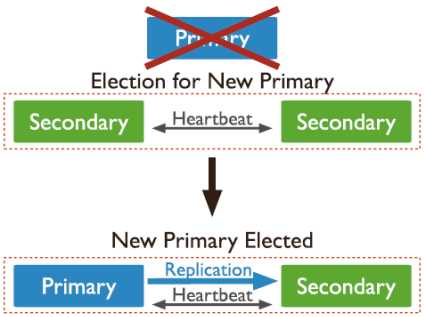
The secondary nodes and replicas

We can have as many secondary nodes as we need.
Some of the secondary nodes can be act like replicas of the primary node.

The Heartbeat approach

It is a method to check if nodes are alive.
The heartbeat is the time that nodes wait before realizing that a certain node is dead.





If the primary node is negative to the heartbeat, it means that it's dead, so other nodes will elect a new primary node.
The data inside the primary node are fully recovered thanks to replicas.

Last Update: Thursday, 22 February 2024

Write concern specification

Write concern can include the following fields:

```
{ w: <value>, j: <boolean>, wtimeout: <number> }
```

- the w option to request acknowledgment that the write operation has propagated to a specified number of mongod instances or to mongod instances with specified tags.
- the j option to request acknowledgment that the write operation has been written to the on-disk journal, and
- the wtimeout option to specify a time limit to prevent write operations from blocking indefinitely.

J Journal log

The journal is the log of all writes operations.
If a write operation includes `j : true`, then it will be registered in the journal.

Acknowledgment behaviour

The w option and the j option determine when mongod instances acknowledge write operations.

Standalone

A standalone mongod acknowledges a write operation either after applying the write in memory or after writing to the on-disk journal. The following table lists the acknowledgment behavior for a standalone and the relevant write concerns:

	j is unspecified	j:true	j:false
w: 1	In memory	On-disk journal	In memory
w: "majority"	On-disk journal *if running with journaling*	On-disk journal	In memory

Last Update: Thursday, 22 February 2024

Replica sets

The value specified to w determines the number of replica set members that must acknowledge the write before returning success. For each eligible replica set member, the j option determines whether the member acknowledges writes after applying the write operation in memory or after writing to the on-disk journal.

`w: "majority"`
Any data-bearing voting member of the replica set can contribute to write acknowledgment of `"majority"` write operations.

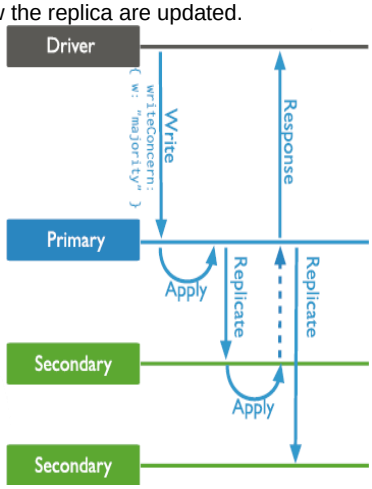
The following table lists when the member can acknowledge the write based on the j value:

`j` is unspecified	Acknowledgment depends on the value of `writeConcernMajorityJournalDefault`: • If `true`, acknowledgment requires writing operation to on-disk journal (`j: true`). `writeConcernMajorityJournalDefault` defaults to `true` • If `false`, acknowledgment requires writing operation in memory (`j: false`).
`j: true`	Acknowledgment requires writing operation to on-disk journal.
`j: false`	Acknowledgment requires writing operation in memory.

Write Concern Type	Behavior
ACKNOWLEDGED	Use the default Write Concern from the server
JOURNALED	Wait for the server to group commit to the journal file on disk.
MAJORITY	Wait on a majority of servers for the write operation.
UNACKNOWLEDGED	Return as soon as the message is written to the socket.
W1	Wait for acknowledgement from a single member.
W2	Wait for acknowledgement from two members
W3	Write operations that use this write concern will wait for acknowledgement from three members

Replication update mechanism

At the right we can see how the replica are updated.



Hierarchy of the write concerns

By default, write concern is organized in a hierarchy, it is global, we can overwrite by specifying a more stringent write concern, both for a given database or for a single collection.

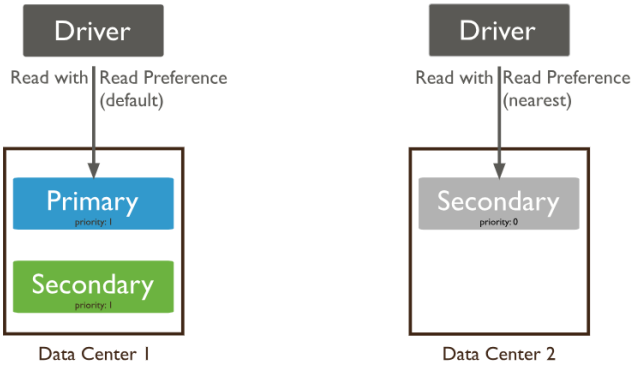
```
// Write concern at client level
MongoClient mongoClient = MongoClients.create(
        "mongodb://localhost:27018,localhost:27019,localhost:27020/" +
                "?w=2&wtimeout=5000");

// Write concern at DB level
MongoDatabase db = mongoClient.getDatabase( s: "LSMDB")
        .withWriteConcern(WriteConcern.W2);

// Write concern at collection level
MongoCollection<Document> myColl = db.getCollection( s: "students")
        .withWriteConcern(WriteConcern.W2);
```

By default, the read preference is configured as primary value, together with write operations are forwarded to the primary node of the replica set.

Last Update: Thursday, 22 February 2024

Read preference

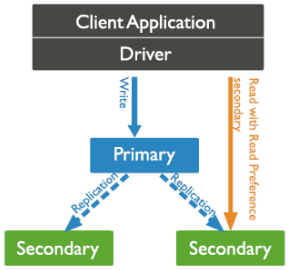
Read preference describes how MongoDB clients route read operations to the members of a replica set.



Behavior of the read preference

All read preference modes except `primary` may return stale data because secondaries replicate operations from the primary in an asynchronous process. [1] Ensure that your application can tolerate stale data if you choose to use a non-`primary` mode.

A **read preference** provides client applications with control over which nodes of a replica set are used for reads



Read Preferences	Behavior
PRIMARY	The default read mode. Read from primary only
PRIMARY PREFERRED	Read from primary if available, otherwise a secondary
SECONDARY	Read from a secondary node if available, otherwise error.
SECONDARY PREFERRED	Read from a secondary if available, otherwise read from the primary.
NEAREST	Read from any member node from the set of nodes which respond the fastest. (Responsiveness measured in pings)

Hierarchy of the read preference

```
// Read Preferences at client level
MongoClient mongoClient = MongoClients.create(
        "mongodb://localhost:27018,localhost:27019,localhost:27020/" +
                "?readPreference=secondary");

// Read Preferences at DB level
MongoDatabase db = mongoClient.getDatabase( s: "LSMDB")
        .withReadPreference(ReadPreference.secondary());

// Read Preferences at collection level
MongoCollection<Document> myColl = db.getCollection( s: "students")
        .withReadPreference(ReadPreference.secondary());

// Tagged read preferences
Tag west_tag = new Tag( name: "dc",  value: "west");
ReadPreference tagged_pref =
        ReadPreference.secondaryPreferred(new TagSet(west_tag));
MongoCollection<Document> myColl = db.getCollection( s: "students")
        .withReadPreference(tagged_pref);

myColl.find().forEach(printDocuments());
```

Last Update: Thursday, 22 February 2024

Replication on MongoDB - shell commands

Let's suppose that we want to create 3 replicas.

Create directories

The first thing to do is to create 3 directories, one for each replica.

```
# Ubuntu
mkdir ~/data/r{1,2,3}


# Windows (PowerShell)
mkdir C:\MongoDB\data\r1
mkdir C:\MongoDB\data\r2
mkdir C:\MongoDB\data\r3
```

Start multiple MongoDB Servers

Then you must start the 3 server processes.

```
# Windows (For Ubuntu it's the same)
mongod --replSet rs0 --dbpath c:\MongoDB\data\r1
--port 27018 --bind_ip localhost --oplogSize 200

mongod --replSet rs0 --dbpath c:\MongoDB\data\r1
--port 27019 --bind_ip localhost --oplogSize 200

mongod --replSet rs0 --dbpath c:\MongoDB\data\r1
--port 27020 --bind_ip localhost --oplogSize 200
```

With `-replSet`, i say that these server processes are replicas.

Connect with mongosh

```
# Windows (For Ubuntu it's the same)
mongosh --port 27017
```

Configure replicas

Define the following configuration (*copy-paste the following code within the mongo shell*).

```
rsconf = {
    _id: "rs0",
    members: [
             {_id: 0, host: "localhost:27018"},
             {_id: 1, host: "localhost:27019"},
             {_id: 2, host: "localhost:27020"}]};
```

Finally

5. Initiate the replica set (only ONCE)

```
rs.initiate(rsconf);
```

6. Check the status:

```
rs.status();
```

7. Change configuration :

```
rs.reconfig(rsconfig);
```

- Set different priorities to replica set members in order to change the default behavior during the primary election

```
{_id: 0, host: "localhost:27018", priority: 0.5}
{_id: 1, host: "localhost:27019", priority: 1},
{_id: 2, host: "localhost:27020", priority: 3}]
```

- Set tags for different members:

```
{_id:0, host:"localhost:27018", tags:{dc:"east"}},
{_id:1, host:"localhost:27019", tags:{dc:"mid"}},
{_id:2, host:"localhost:27020", tags:{dc:"west"}}]
```

```
rsconf = {
    _id: "lsmdb",
    members: […],
    settings: {
       setDefaultRWConcern :{w: 2, wtimeout : 5000}}};
```

- **W** is the minimum *number of writes* that the application waits until it regains the flow of execution. The default value is *majority*, but it can be any integer >= 1.
- **Wtimeout** is the maximum *waiting time* (in *ms*) before regaining the flow of execution. If set to 0 the application will wait indefinitely or until "w" is satisfied.

```
// Method 1
MongoClient mongoClient = MongoClients.create(
        "mongodb://localhost:27018,localhost:27019,localhost:27020/" +
                "?retryWrites=true&w=majority&wtimeout=10000");

// Method 2
// You can specify one or more nodes in the replica set.
// MongoDB will automatically discover PRIMARY and SECONDARY nodes within the cluster
uri = new ConnectionString("mongodb://localhost:27018");
MongoClientSettings mcs = MongoClientSettings.builder()
        .applyConnectionString(uri)
        .readPreference(ReadPreference.nearest())
        .retryWrites(true)
        .writeConcern(WriteConcern.ACKNOWLEDGED).build();
MongoClient mongoClient2 = MongoClients.create(mcs);
```

- In this tutorial we have, at our disposal, the following **VMs**

VM	IP address	OS
Replica-0	10.1.1.10	Ubuntu 20.04.04
Replica-1	10.1.1.11	Ubuntu 20.04.04
Replica-2	10.1.1.12	Ubuntu 20.04.04

- First, we need to connect through a VPN
- Install mongoDB on each VM
- Start and configure the servers in a similar fashion as before

Last Update: Thursday, 22 February 2024

Follow these steps to install MongoDB Community Edition using the `apt` package manager.

1 Import the public key used by the package management system

From a terminal, install `gnupg` and `curl` if they are not already available:

```
sudo apt-get install gnupg curl
```

To import the MongoDB public GPG key from https://pgp.mongodb.com/server-7.0.asc, run the following command:

```
curl -fsSL https://pgp.mongodb.com/server-7.0.asc | \
   sudo gpg -o /usr/share/keyrings/mongodb-server-7.0.gpg \
   --dearmor
```

2 Create a list file for MongoDB

Create the list file `/etc/apt/sources.list.d/mongodb-org-7.0.list` for your version of Ubuntu.

Ubuntu 22.04 (Jammy) **Ubuntu 20.04 (Focal)**

Create the `/etc/apt/sources.list.d/mongodb-org-7.0.list` file for Ubuntu 20.04 (Focal):

```
echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-7.
```

3 Reload local package database

Issue the following command to reload the local package database:

```
sudo apt-get update
```

4 Install the MongoDB packages

You can install either the latest stable version of MongoDB or a specific version of MongoDB.

Install the latest version of MongoDB Install a specific release of MongoDB

To install the latest stable version, issue the following

```
sudo apt-get install -y mongodb-org
```

Large-Scale VPN

1. Install OpenVPN from https://openvpn.net/community-downloads/
2. Download the *largescale.ovpn* file containing all the parameters for the VPN.
3. Start OpenVPN and select "Import from file"
4. Select the *largescale.ovpn* file and start the connection

If everything goes well, you should be able to connect through ssh via user=**root** and pass=**root.**

```
ssh root@10.1.1.10
```

- Open 3 different terminals and issues the following commands

```
#Terminal 1
ssh root@10.1.1.10
mkdir data
mongod --replSet lsmdb --dbpath ~/data --port
27020 --bind_ip localhost,172.16.4.99 --oplogSize
200
```

```
#Terminal 2
ssh root@10.1.1.11
mkdir data
mongod --replSet lsmdb --dbpath ~/data --port
27020 --bind_ip localhost,172.16.4.100 --oplogSize
200
```

```
#Terminal 3
ssh root@10.1.1.12
mkdir data
mongod --replSet lsmdb --dbpath ~/data --port
27020 --bind_ip localhost,172.16.4.101 --oplogSize
200
```

Last Update: Thursday, 22 February 2024

- Connect to one of the VM through ssh (open a 4th terminal)

```
#Terminal 4
ssh root@172.16.4.99
mongo --port 27020

rsconf = {
    _id: "lsmdb",
    members: [
        {_id: 0, host: "10.1.1.10:27020", priority:1},
        {_id: 1, host: "10.1.1.11:27020", priority:2},
        {_id: 2, host: "10.1.1.12:27020", priority:5}]
};

rs.initiate(rsconf);
```