BDM-MIRI - Big Data Management

Jose Antonio Lorencio Abril

Spring 2023

Professor: Alberto Abelló

Student e-mail: jose.antonio.lorencio@estudiantat..upc.edu

This is a summary of the course *Big Data Management* taught at the Universitat Politècnica de Catalunya by Professor Alberto Abelló in the academic year 22/23. Most of the content of this document is adapted from the course notes by Abelló and Nadal, [1], so I won't be citing it all the time. Other references will be provided when used.

Contents

List of Figures

List of Tables

List of Algorithms

1   Introduction to Big Data

1.1   Recognise the relevance of data driven decision making

Data driven decision making is the strategy of using data to make decisions, in order to improve the chances of obtaining a positive outcome. It has been gaining importance in the past years, mainly because the data generation rate is increasing rapidly, allowing greater analyses for those who are able to leverage all this data.

The ability to collect, store, combine and analyze relevant data enables companies to gain a competitive advantage over their competitors which are not able to take on these task.

In a nutshell, it is the confluence of three major socio-economic and technological trends that makes data driven innovation a new phenomenon:

- The exponential growth in data generated and collected.

- The widespread use of data analytitcs, including start-ups and small and medium entreprises.

- The emergence of a paradigm shift in knowledge.

1.2   Identify the three high level categories of analytical tools

Business Intelligence (BI) is the concept of using dashboard to represent the status and evolution of companies, using data from the different applications used by the production systems of the company, which needs to be processed with ETL (Extract, Transform, Load) pipelines into a Data Warehouse. This data is then modelled into data cubes, that are queried with OLAP (OnLine Analytic Processing) purposes. The analytical tools that this setup allows are three:

1. Static generation of reports.

2. Dynamic (dis)aggregation and navigation by means of OLAP operations.

3. Inference of hidden patterns or trends with data mining tools.

1.3   Identify the two main sources of Big Data

The two main sources of Big Data are:

- The **Internet**, which shifted from a passive role, where static hand-crafted contents were provided by some gurus, to a dynamic role, where contents can be easily generated by anybody in the world, specially through social networks.

- The **improvement of automation and digitalization** on the side of industries, which allows to monitor many relevant aspects of the company's scope, giving rise to the concept of Internet of Things (IoT), and generating a continuous flow of information.

1.4   Give a definition of Big Data

Big Data is a natural evolution of Business Intelligence, and inherits its ultimate goal of transforming raw data into valuable knowledge, and it can be characterized in terms of the **five V's**:

- **Volume**: there are large amount of digital information produced and stored in new systems.

- **Velocity**: the pace at which data is generated, ingested and processed is very fast, giving rise to the concept of data stream (and two related challenges: *data stream ingestion* and *data stream processing*).

- **Variety**: there are multiple, heterogeneous data formats and schemas, which need to be dealt with. Special attention is needed for semi-structured and unstructured external data. The *data variety challenge* is considered as the most crucial challenge in data driven organizations.

- **Variability**: the incoming data can have an evolving nature, which the system needs to be able to cope with.

- **Veracity**: the veracity of the data is related to its quality, and it makes it compulsory to develop *Data Governance* practices, to effectively manage data assets.

1.5 Compare traditional data warehousing against Big Data management

In traditional business intelligence, data from different sources inside the company is ETL-processed into the data warehouse, which can then be analyzed using the three types of analyses we've seen (Reports, OLAP, DM), in order to extract useful information that ultimately affects the strategy of the company. This is summarized in Figure 1. As highlighted in the figure, the data warehousing process encompasses the ETL processes and the Data Warehouse design and maintenance.
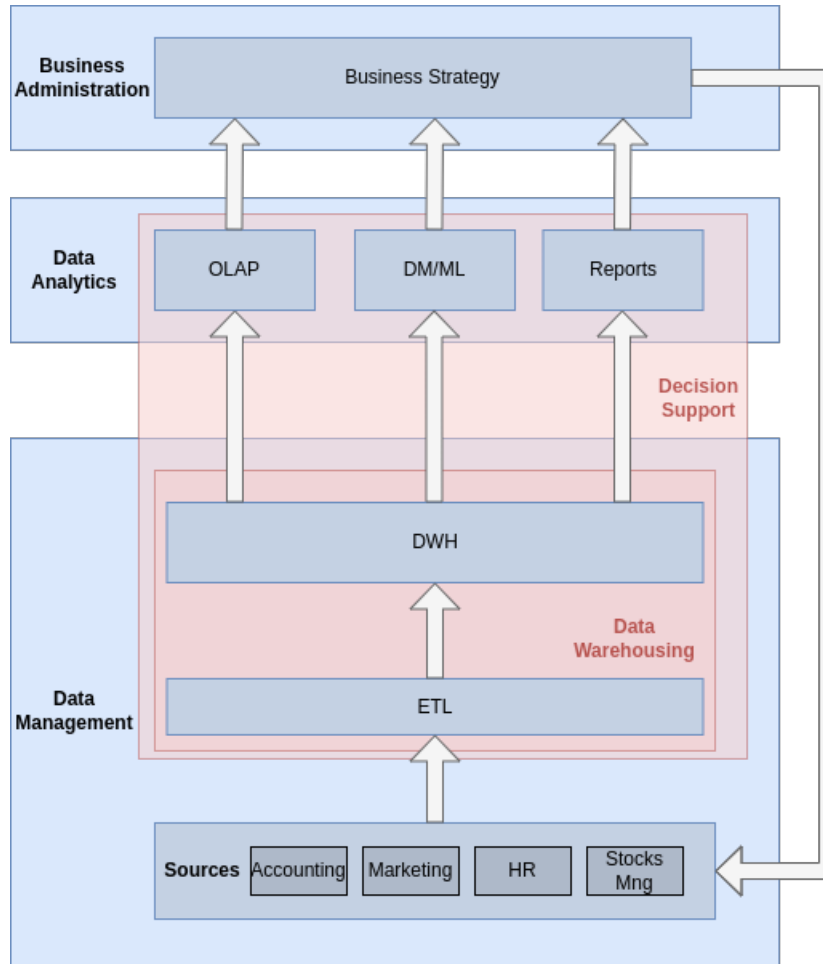
Figure 1: Business Intelligence Cycle.

In the context of big data, the focus is shifted, from analyzing data from just inside sources, to data from all types of heterogeneous sources. In this setup, instead of doing an ETL process, the data is collected, through the process of ingestion, and stored into a Data Lake (from which analysts would extract data and perform all necessary transformations a posteriori) or a Polystore (which is a DBMS built on top of different other technologies, to be able to cope with heterogeneous data). Whatever the storing decision, Big Data Analytics are then done on this data, differenciating:

- Small analytics: querying and reporting the data and OLAP processing.

- Big Analytics: performing data mining on the data.

This process is depicted in Figure 2. In this diagram, we see that the **Big Data Management** consists of the task of ingestion, together with the design and maintenance of the Data Lake / Polystore.
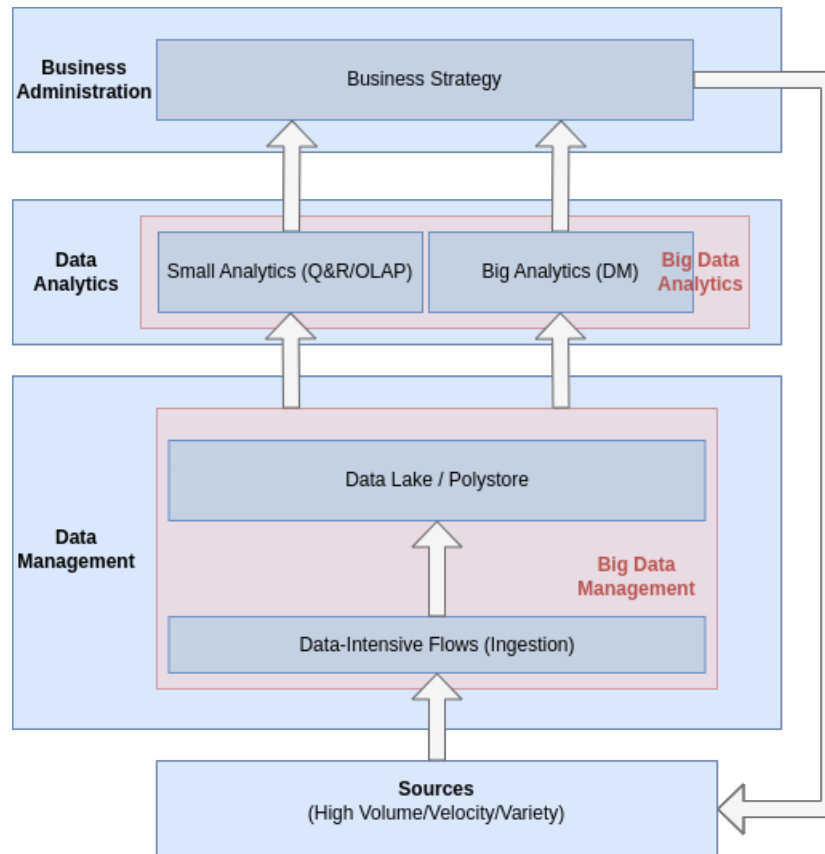
Figure 2: Big Data Cycle.

Thus, the differences are:

- Data Warehousing does the ETL process over the data produced by the company, while Big Data Management does the process of ingestion, by which data from internal and external sources is collected.

- Data Warehousing uses a Data Warehouse to store the ETLed data and the analyses need to be designed with the structure of this stored data. In contrast, in Big Data Management, the storing facility can cope with the data as is, so the analyses have a wider scope, but they need to correctly treat the data for each analysis conducted.

- Thus, as can be inferred from the previous paragraphs, Big Data Management provides a more flexible setup than Data Warehousing, at the expense of needing to perform ad-hoc transformation for each analysis, which can lead to repetition and a decrease in performance. Nonetheless, this decrease is not really a drawback, because some big data analytics tasks cannot be undertaken without this added flexibility.

1.6   Distinguish descriptive, predictive and prescriptive analysis

- **Descriptive analysis**: uses basic statistics to describe the data. In a DW environment, OLAP tools are used for this purpose, in an interactively manner, modifying the analysis point of vire to facilitate the understanding and gain knowledge about the stored data. Basically, understand past data (what happened, when happened, why it happened).

- **Predictive analysis**: uses a set of statistical techniques to analyze historical facts, with the aim of making predictions about future events. Basically, compare incoming data to our knowledge of past data, in order to make predictions about the future (what will happen).

- **Prescriptive analysis**: takes as input the predictions of previous analyses to suggest actions, decisions and describe the possible implications of each of them. Basically, use predictions obtained via predictive

analysis to take action and make decisions, as well as to estimate the impact of these decisions in the future (how we should respond to this situation).

1.7 Explain the novelty of Cloud Computing

The novelty of cloud computing is the same as when electricity shifted from being generated by each company to be centrally generated, benefiting from scale economies and improving the efficiency of the electricity generation. In the case of Cloud Computing, the shift is from companies having their own hardware and software, to an environment in which these resources are offered by a third company, which leverages again the economies of scale and the possibility to allocate resources when needed, increasing the overall efficiency of the tech industries and reducing the costs of each company, as they now don't need to buy expensive pieces of hardware and software, maintain them, etc.

1.8 Justify the benefits of Cloud Computing

- It eliminates upfront investment, as it is not needed to buy hardware anymore.

- You pay for what you use, so costs are reduced because efficient allocation is a complex task to overcome.

- The main benefit comes from the aforementioned economy of scale, that allows to reduce costs and improve efficiency. A machine hosted in-house is most of the time underused, because companies don't usually require it being 100% operational all the time. However, when the machine is available for thousand or millions of customers, it will almost always be required to be working.

- Customers can adapt their costs to their needs at any time.

- There is no need to manage, maintain and upgrade hardware anymore.

1.9 Explain the link between Big Data and Cloud Computing

Cloud computing and big data are closely related, and in many ways, cloud computing has enabled the growth and adoption of big data technologies.

One of the main advantages of cloud computing is its ability to provide flexible and scalable computing resources on demand. This is especially important for big data, which requires significant computing power to process and analyze large volumes of data. Cloud computing allows organizations to easily spin up large-scale computing clusters and storage systems to handle big data workloads, without the need to invest in expensive on-premises infrastructure.

In addition to providing scalable computing resources, cloud computing also offers a wide range of data storage and processing services that can be used for big data workloads. Cloud providers offer a variety of data storage services, such as object storage, file storage, and database services, that can be used to store and manage large volumes of data. Cloud providers also offer big data processing services, such as Apache Hadoop, Apache Spark, and machine learning tools, which can be used to analyze and extract insights from big data.

Cloud computing also provides the ability to easily integrate and share data between different systems and applications, both within an organization and with external partners. This is important for big data, which often requires data from multiple sources to be combined and analyzed to gain insights.

Overall, cloud computing has played a key role in enabling the growth and adoption of big data technologies, by providing flexible and scalable computing resources, a wide range of data storage and processing services, and the ability to easily integrate and share data between different systems and applications.

1.10 Distinguish the main four service levels in Cloud Computing

The main four service levels are:

- **Infrastructure as a Service (IaaS)**: provides virtualized computing resources, such as virtual machines, storage, and networking, which can be provisioned and managed through an API or web console.

- **Platform as a Service (PaaS)**: provides a platform for building and deploying applications, including development tools, runtime environments, and middleware, which can be accessed through an API or web console.

- **Software as a Service (SaaS)**: provides access to software applications over the internet, which are hosted and managed by a third-party provider, and can be accessed through a web browser or API.

- **Business as a Service (BaaS)**: This is a type of cloud computing service that provides businesses with access to a range of software tools and services, such as customer relationship management (CRM) systems, enterprise resource planning (ERP) software, and human resources management tools. BaaS allows businesses to outsource the management and maintenance of these systems to a third-party provider, freeing up resources and allowing the business to focus on their core operations. BaaS can be a cost-effective way for businesses to access enterprise-level software tools without the need to invest in on-premises infrastructure and maintenance. This is, a whole business process is outsourced, for example using PayPal as a paying platform frees the company from this process.

But there are more services offered by Cloud Computing:

- **Database as a Service (DBaaS)**: specific platform services providing data management functionalities.

- **Container as a Service (CaaS)**: allows applications to be packaged into containers, which can be run consistently across different environments, such as development, testing, and production.

- **Function as a Service (FaaS)**: creates small stand-alone pieces of software that can be easily combined to create business flows in interaction with other pieces from potentially other service providers.

- **Serverless computing**: allows developers to build and run applications without managing servers, by providing an event-driven computing model, in which code is executed in response to specific triggers.

- **Data analytics and storage**: provides tools for storing and analyzing large volumes of data, such as data warehouses, data lakes, and analytics tools, which can be accessed through APIs or web consoles.

- **Machine learning and artificial intelligence**: Provides tools and services for building, training, and deploying machine learning models, such as pre-trained models, APIs for image recognition and natural language processing, and tools for custom model development.

2 Big Data Design

2.1 Define the impedance mismatch

Impedance mismatch often arises when data is passed between different layers of an application, such as between the front-end user interface and the back-end database, or between different applications that need to exchange data. The data structures used in each layer or system may be different, which can cause issues with data mapping, performance, and scalability.

For example, if a front-end application requires data that is stored in a relational database, the application may need to perform complex queries to retrieve and transform the data into a format that can be used by the user interface. This can lead to performance issues and increased complexity in the application code. Similarly, if different applications or services use different data formats or structures, it can be difficult to exchange data between them, which can lead to integration issues and increased development time.

To address impedance mismatch, software developers often use techniques such as object-relational mapping (ORM) to map data between different layers of an application, or use standard data formats such as JSON or XML to enable data exchange between different systems. These techniques can help to simplify data mapping, improve performance, and increase the scalability of the system.

2.2 Identify applications handling different kinds of data

- **Relational data (OLTP)**: Relational databases are commonly used for online transaction processing (OLTP) applications, such as e-commerce websites, banking applications, and inventory management systems. Examples of applications that use relational databases include Oracle, MySQL, PostgreSQL, and Microsoft SQL Server.

- **Multidimensional data (OLAP)**: Multidimensional databases are commonly used for online analytical processing (OLAP) applications, such as data warehousing, business intelligence, and data mining. Examples of applications that use multidimensional databases include Microsoft Analysis Services, IBM Cognos, and Oracle Essbase.

- **Key-value data**: Key-value databases are commonly used for high-performance, highly scalable applications, such as caching, session storage, and user profiles. Examples of applications that use key-value databases include Redis, Amazon DynamoDB, and Apache Cassandra.

- **Column-family data**: Column-family databases are commonly used for applications that require fast reads and writes on a large-scale, such as content management systems, social networks, and recommendation engines. Examples of applications that use column-family databases include Apache HBase, Apache Cassandra, and ScyllaDB.

- **Graph data**: Graph databases are commonly used for applications that involve complex relationships between data, such as social networks, fraud detection, and recommendation engines. Examples of applications that use graph databases include Neo4j, OrientDB, and Amazon Neptune.

- **Document data**: Document databases are commonly used for applications that require flexible, dynamic data structures, such as content management systems, e-commerce platforms, and mobile applications. Examples of applications that use document databases include MongoDB, Couchbase, and Amazon DocumentDB.

Note that many applications use multiple types of data models, depending on the nature of the data and the requirements of the application. For example, a social network might use a graph database to store social connections, a column-family database to store user data, and a key-value database to cache frequently accessed data.

2.3 Name four different kinds of NOSQL systems

- **Key-Value**: stores data as a collection of key-value pairs. Each key is associated with a value, and values can be retrieved and updated by their corresponding keys. Key-value databases are simple and highly scalable, making them well-suited for applications that require high performance and low latency.

- **Wide-column (Column-family)**: stores data as a collection of columns grouped into column families. Each column family is a group of related columns, and each column consists of a name, a value, and a timestamp. Column-family databases are optimized for storing large amounts of data with fast writes and queries, making them well-suited for applications that require high write and query throughput. Such grouping of columns directly translates into a vertical partition of the table, and entails the consequent loss of schema flexibility.

- **Graph**: stores data as nodes and edges, representing complex relationships between data. Nodes represent entities, such as people, places, or things, and edges represent relationships between entities. Graph databases are optimized for querying and analyzing relationships between data, making them well-suited for applications that require complex querying.

- **Document**: stores data as documents, which can be thought of as semi-structured data with a flexible schema. Each document consists of key-value pairs, and documents can be grouped into collections. Document stores are optimized for storing and querying unstructured and semi-structured data, making them well-suited for applications that require flexibility in data modeling.

2.4 Explain three consequences of schema variability

Schema variability refers to the dynamic and flexible nature of data models in NoSQL databases. Unlike relational databases, NoSQL databases allow for the schema to be flexible and adaptable, which means that the data structure can evolve over time without requiring changes to the database schema. This allows for greater agility in data modeling, as it makes it easier to add or remove fields or change the data structure as needed.

Its three main consequences are:

- **Gain in flexibility**: allowing schema variability makes the system more flexible to cope with changes in the data.

- **Reduced data semantics and consistency**: With a flexible schema, it is possible to store data that does not conform to a predefined structure. This can lead to inconsistencies in data quality and make it more difficult to enforce data constraints, such as data types or referential integrity.

- **Data independence principle is lost**: allowing schema variability can be seen as a departure from the traditional concept of data independence, which is a key principle of the relational data model. Data independence refers to the ability to change the physical storage or logical structure of the data without affecting the application programs that use the data. In a relational database, the data is organized into tables with fixed schema, which allows for greater data independence.

  However, in NoSQL databases, schema variability is often seen as a necessary trade-off for achieving greater flexibility and scalability. By allowing for a more flexible data model, NoSQL databases can better accommodate changes to the data structure over time, without requiring changes to the database schema or application code. This can help improve agility and reduce development time.

  That being said, NoSQL databases still adhere to the fundamental principles of data independence in many ways. For example, they still provide a layer of abstraction between the application and the physical storage of the data, which helps to insulate the application from changes to the underlying data storage. Additionally, many NoSQL databases provide APIs that allow for flexible querying and manipulation of data, which helps to maintain a level of data independence.

Some more consequences:

- Increased data complexity: As the data model becomes more flexible, the data can become more complex and difficult to manage. This can lead to increased development and maintenance costs, as well as potential performance issues.

- Increased development and maintenance costs: As the schema becomes more flexible, the complexity of the data model can increase, which can result in higher development and maintenance costs.

- Reduced performance: With a more complex data model, queries can become more complex, which can result in slower query performance. Additionally, since the schema is not fixed, indexing and optimization become more difficult, which can further impact performance.

2.5 Explain the consequences of physical independence

Physical independence is a key principle of the relational data model, which refers to the ability to change the physical storage of the data without affecting the logical structure of the data or the application programs that use the data. This means that the application should be able to access and manipulate the data without being aware of the underlying physical storage details, such as the storage medium or the location of the data.

The consequences of physical independence include:

- Reduced maintenance costs: With physical independence, it is easier to change the physical storage of the data without affecting the application. This can help reduce maintenance costs, as it allows for more flexibility in how the data is stored and accessed over time.

- Improved scalability: Physical independence can help improve scalability, as it allows for the data to be distributed across multiple physical storage locations or devices, which can help to improve performance and reduce the impact of failures.

- Greater portability: With physical independence, the application is not tied to a specific physical storage medium or location, which can help improve portability across different hardware or software platforms.

- Improved performance: Physical independence can help improve performance, as it allows for the data to be stored and accessed in the most efficient way possible, without being limited by the constraints of a specific physical storage medium or location.

Nonetheless, physical independence enhance the problem of the impedance mismatch, because if data is needed in a different form from how it is stored, it has to be transformed, introducing a computing overhead. If we store the data as needed for the application, this problem is reduced, but the physical independence can be lost.

2.6 Explain the two dimensions to classify NOSQL systems according to how they manage the schema

The schema can be explicit/implicit:

- **Implicit schema**: schema that is not explicitly defined or documented. Instead, the schema is inferred or derived from the data itself, usually through analysis or observation. This can be useful in situations where the data is very dynamic or unstructured, and where the structure of the data is not known in advance.

- **Explicit schema**: schema that is explicitly defined and documented, usually using a schema language or a data modeling tool. The schema specifies the types of data that can be stored, the relationships between different types of data, and any constraints or rules that govern the data.

And it can be fixed/variable:

- **Fixed schema**: schema that is static and unchanging, meaning that the structure and organization of the data is predefined and cannot be modified. This is common in relational databases, where the schema is usually defined in advance and remains fixed over time.

- **Variable schema**: schema that is dynamic and flexible, meaning that the structure and organization of the data can change over time. This is common in NoSQL databases, where the schema may be more fluid and adaptable to changing data requirements.

Note that this is not a strict classification, but rather two dimension ranges in which a certain schema can lie.

2.7 Explain the three elements of the RUM conjecture

The RUM conjecture suggests that in any database system, the overall performance can be characterized by a trade-off between the amount of memory used, the number of reads performed, and the number of updates performed. Specifically, the conjecture states that there is a fundamental asymmetry between reads and updates, and that the performance of the system is strongly influenced by the balance between these two operations. In general, the more reads a system performs, the more memory it requires, while the more updates it performs, the more it impacts the system's overall performance.

The RAM conjecture has three main elements:

- Reads: refer to the process of retrieving data from a database. In general, read-heavy workloads require more memory to achieve good performance.

- Updates: refer to the process of modifying data in a database. In general, update-heavy workloads require more processing power and can negatively impact overall performance.

- Memory: refers to the amount of memory available to a system. In general, increasing memory can improve read-heavy workloads, but may not be as effective for update-heavy workloads.

The RUM conjecture is often used to guide the design and optimization of database systems, as it provides a useful framework for understanding the trade-offs between different system parameters and performance metrics. By understanding the RUM trade-offs, database designers can make informed decisions about how to allocate resources, optimize queries, and balance the workload of the system.

2.8   Justify the need of polyglot persistence

A **polyglot system** is a system that uses multiple technologies, languages, and tools to solve a problem. In the context of data management, a polyglot system is one that uses multiple data storage technologies to store and manage data. For example, a polyglot system might use a combination of relational databases, NoSQL databases, and search engines to store different types of data.

   Polyglot persistence is the practice of using multiple storage technologies to store different types of data within a single application. The idea is to choose the right tool for the job, and to use each technology to its fullest potential. For example, a polyglot system might use a NoSQL database to store unstructured data, a relational database to store structured data, and a search engine to provide full-text search capabilities.

   There are several reasons why polyglot persistence is important:

- **Flexibility**: Polyglot systems are more flexible than monolithic systems that use a single technology to store all data. With a polyglot system, you can choose the right tool for the job, and you can adapt to changing requirements and data formats.

- **Performance**: Different data storage technologies are optimized for different types of data and workloads. By using the right tool for the job, you can improve performance and scalability.

- **Resilience**: Using multiple data storage technologies can improve the resilience of your system. If one database fails, the other databases can continue to operate, ensuring that your application remains available and responsive.

- **Future-proofing**: By using multiple data storage technologies, you can future-proof your system against changing data formats and requirements. As new data types and storage technologies emerge, you can add them to your system without having to completely overhaul your architecture.

In summary, polyglot persistence is a powerful approach to data management that allows you to use multiple storage technologies to store and manage different types of data within a single application. By adopting a polyglot approach, you can improve flexibility, performance, resilience, and future-proofing.

2.9   Decide whether two NOSQL designs have more or less explicit/fix schema

There are several factors that can be used to assess the flexibility and explicitness of a schema:

1. Number of tables/collections: A schema with a large number of tables or collections is typically more explicit and less flexible than a schema with fewer tables or collections. This is because a large number of tables or collections often implies a more rigid structure, whereas a smaller number of tables or collections can allow for more flexibility.

2. Number of columns/fields: A schema with a large number of columns or fields is typically more explicit and less flexible than a schema with fewer columns or fields. This is because a large number of columns or fields often implies a more rigid structure, whereas a smaller number of columns or fields can allow for more flexibility.

3. Data types: A schema that uses a large number of data types is typically more explicit and less flexible than a schema that uses fewer data types. This is because a large number of data types often implies a more rigid structure, whereas a smaller number of data types can allow for more flexibility.

4. Use of constraints: A schema that uses a large number of constraints (such as foreign keys or unique constraints) is typically more explicit and less flexible than a schema that uses fewer constraints. This is because constraints often imply a more rigid structure, whereas a schema with fewer constraints can allow for more flexibility.

5. Use of inheritance: A schema that uses inheritance (such as table or collection inheritance) is typically more flexible and less explicit than a schema that does not use inheritance. This is because inheritance allows for more flexibility in the structure of the data, whereas a schema that does not use inheritance is typically more explicit in its structure.

Overall, a more explicit schema is one that has a more rigid structure, with more tables, fields, data types, and constraints, whereas a more flexible schema is one that has fewer tables, fields, data types, and constraints, and may use inheritance to provide more flexibility.

Some examples can be:

- Fixed/Explicit: An example of a fixed/explicit schema in XML format might look like this:

```
1  <shopping_cart>
2    <customers>
3      <customer>
4        <name>John Smith</name>
5        <email>john@example.com</email>
6      </customer>
7    </customers>
8    <orders>
9      <order>
10       <order_date>2023-02-17</order_date>
11       <total_price>100.00</total_price>
12     </order>
13   </orders>
14   <products>
15     <product>
16       <name>Widget</name>
17       <description>A small, useful tool</description>
18       <price>10.00</price>
19     </product>
20   </products>
21 </shopping_cart>
```

  In this example, the schema is fixed because there are specific tables (customers, orders, and products) and specific fields for each table (such as name and email for customers, and order_date and total_price for orders). There is no room for variation in the structure of the schema.

- Fixed/Implicit: An example of a fixed/implicit schema in XML format might look like this:

```
1  <blog>
2    <posts>
3      <post>
4        <title>My First Blog Post</title>
5        <content>This is my first blog post.</content>
6      </post>
7    </posts>
8  </blog>
```

  In this example, the schema is fixed because there is a specific table (posts) and specific fields for that table (such as title and content). However, there is no fixed field for metadata such as tags or categories.

- Flexible/Explicit: An example of a flexible/explicit schema in XML format might look like this:

```
1  <scientific_data>
2    <experiments>
3      <experiment>
```

```
4        <date >2023 -02 -17 </date >
5        <sample_size >100 </ sample_size >
6        <measurement_units >mg/L</ measurement_units >
7      </experiment >
8      <experiment >
9        <date >2023 -02 -16 </date >
10        <sample_size >50</ sample_size >
11        <measurement_units >g/L</ measurement_units >
12      </experiment >
13    </ experiments >
14    <observations >
15      <observation >
16        <value >10.00 </value >
17      </observation >
18      <observation >
19        <value >20.00 </value >
20      </observation >
21    </ observations >
22  </scientific_data >
```

In this example, the schema is flexible because there can be any number of experiments and observations, and there are no fixed fields for metadata. However, each table (experiments and observations) and each field (such as date and sample_size) is explicitly defined in the schema.

- Flexible/Implicit: An example of a flexible/implicit schema in XML format might look like this:

```
1  <social_media >
2    <posts >
3      <post >
4        <text >Hello world!</text >
5      </post >
6    </posts >
7    <comments >
8      <comment >
9        <text >Great post!</text >
10      </comment >
11      <comment >
12        <text >Thanks for sharing.</text >
13      </comment >
14    </comments >
15  </social_media >
```

In this example, the schema is flexible because there can be any number of posts and comments, and there are no fixed fields for any table. The structure of the schema is also implicit because there is no fixed structure for the data.

2.10   Given a relatively small UML conceptual diagram, translate it into a logical representation of data considering flexible schema representation

3 Distributed Data Management

3.1 Give a definition of Distributed System

A distributed system is a system whose components, located at networked computers, communicate and coordinate their actions only by passing messages.

3.2 Enumerate the six challenges of a Distributed System

The challenges of a distributed system are:

- **Scalability**: the system must be able to continuously evolve to support a grouwing amount of tasks. This can be achieve by:

  - Scale up: upgrading or improving the components.
  - Scale out: adding new components.

  Scale out mitigates bottlenecks, but extra communication is needed between a growing number of components. This can be partially solved using direct communication between peers. Load-balancing is also crucial and, ideally, should happen automatically.

- **Performance/efficiency**: the system must guarantee an optimal performance and efficient processing. This is usually measured in terms of latency, response time and throughput. Parallelizing reduces response time, but uses more resources to do it, negatively affecting throughput unless resources are increased to compensate.

  This effect can be mitigated optimizing network usage or using distributed indexes.

- **Reliability and availability**: the system must perform tasks consistently and without failure: it must be **reliable**. It also must keep performing tasks even if some of its components fail: it must be **available**. The availability is not always possible, and some functionalities might be affected, but at least a partial service could be provided when a component fails.

  To increase failure tolerance, **heartbeat mechanisms** can be used to monitor the status of the components, together with **automatic recovery mechanisms**.

  It is also important to keep the consistency of data shared by different components, since this requires synchronization. This can be mitigated by **asynchronous synchronization mechanisms** and **flexible routing of network messages**.

- **Concurrency**: the system should provide the required control mechanisms to avoid interferences and deadlocks in the presence of concurrent requests. **Consensus protocols** can help solving conflicts and enabling the system to keep working without further consequences.

- **Transparency**: users of the system should not be aware of all the aforementioned complexities. Ideally, they should be able to work as if the system was not distributed.

3.3 Give a definition of Distributed Database

A **Distributed Database (DDB)** is an integrated collection of databases that is physically distributed across sites in a computer network and a **Distributed Database Management System (DDBMS)** is the software system that manages a distributed database such that the distribution aspects are transparent to the users.

There are some terms worth detailing:

- **Integrated**: files in the database should be somehow structured, and an access interface common to all of them should be provided so that the physical location of data does not matter.

- **Physically distributed across sites in a computer network**: data may be distributed over large geographical areas but it could also be the case where distributed data is, indeed, in the very same room. The required characteristic is that the communication between nodes is done through a computer network instead of simply sharing memory or disk.

- **Distribution aspects are transparent to the users**: transparency refers to separation of the higher-level view of the system from lower-level implementation issues. Thus, the system must provide mechanisms to hide the implementation details.

3.4 Explain the different transparency layers in DDBMS

In a DDBMS **distribution transparency** must be ensured, i.e., the system must guarantee data, network, fragmentation and replication transparency:

- **Data independence**: data definition occurs at two different levels:

  - Logical data independence: refers to indifference of user applications to changes in the logical structure of the database.
  - Physical data independence: hides the storage details to the user.

- **Network transparency**: the user should be protected from the operation details of the network, even hiding its existence whenever possible. There are two subclasses:

  - Location transparency: any task performed should be independent of both the location and system where the operation must be performed.
  - Naming transparency: each object must have a unique name in the database, irrespectively of its storage site.

- **Replication transparency**: refers to whether synchronizing replicas is left to the user or automatically performed by the system. Ideally, all these issues should be transparent to users, and they should act as if a single copy of data were available.

- **Fragmentation transparency**: when data is fragmented, queries need to be translated from the global query into fragmented queries, handling each fragment. This translation should be performed by the DDBMS, transparently to the user.

Note that all these transparency levels are incremental.

Note also that full transparency makes the management of distributed data very difficult, so it is widely accepted that data independence and network transparency are a must, but replication and/or fragmentation transparency might be relaxed to boost performance.

3.5 Identify the requirements that distribution imposes on the ANSI/SPARC architecture

The **Extended ANSI/SPARC architecture** was designed to provide a comprehensive framework for organizing and managing complex database systems. The extended architecture includes the same three levels as the original ANSI/SPARC architecture, but it adds a fourth level, called the user level.

The four levels of the extended ANSI/SPARC architecture are:

- User Level: The user level is the highest level and includes the end users or applications that access the database system. The user level provides a simplified view of the data that is available in the system, and it defines the interactions between the user and the system.

- External Level: The external level is the next level down and includes the external schemas that define the view of the data that is presented to the end users or applications. Each external schema is specific to a particular user or group of users and provides a simplified view of the data that is relevant to their needs.

- Conceptual Level: The conceptual level is the third level and includes the global conceptual schema that describes the overall logical structure of the database system. The global conceptual schema provides a unified view of the data in the system and defines the relationships between different data elements.

- Internal Level: The internal level is the lowest level and includes the physical schema that defines the storage structures and access methods used to store and retrieve the data. The internal schema is specific to the particular database management system and hardware platform that is being used.
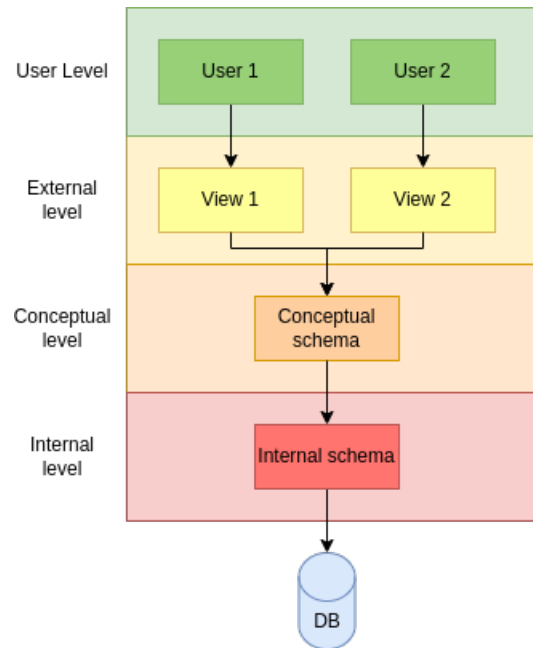
Figure 3: Extended ANSI/SPARC architecture.

This is summarized in Figure 3.

This architecture does not consider distributed, so it need to be consequently adapted to provide distribution trasnparency. To this end, a global conceptual schema is needed to define a single logical database. But the database is composed of several nodes, each of which must now define a local conceptual schema and an internal schema. These adaptations are depicted in Figure 4.

In both architectures, mappings between each layer are stored in the global catalog, but in the distributed architecture there two mappings which are particularly important, namely the **fragmentation schema** and the **allocation schema**.

3.6 Draw a classical reference functional architecture for DDBMS

The functional architecture of a centralized DBMS is depicted in Figure 5. The **query manager** is a component of a database management system (DBMS) that is responsible for handling user queries and managing the overall query processing. It is composed of several sub-components:

- The **view manager** is responsible for managing the views defined in the system. Views are virtual tables that are derived from the base tables in the database and are used to simplify the user's interaction with the database. The view manager translates user queries that reference views into queries that reference the base tables, allowing the user to interact with the database at a higher level of abstraction.

- The **security manager** is responsible for enforcing security policies and access controls in the system. It ensures that only authorized users are allowed to access the database and that they only have access to the data that they are authorized to see. The security manager also enforces constraints and ensures that the data in the database is consistent and valid.

- The **constraint checker** is responsible for verifying that the data in the database conforms to the integrity constraints defined in the schema. It checks for violations of primary key, foreign key, and other constraints, and ensures that the data in the database is consistent and valid.

- The **query optimizer** is responsible for optimizing user queries to improve performance. It analyzes the query and determines the most efficient way to execute it, taking into account factors such as the available indexes, the size of the tables involved, and the cost of different query execution plans. The query optimizer generates an optimal query execution plan that minimizes the time required to process the query.
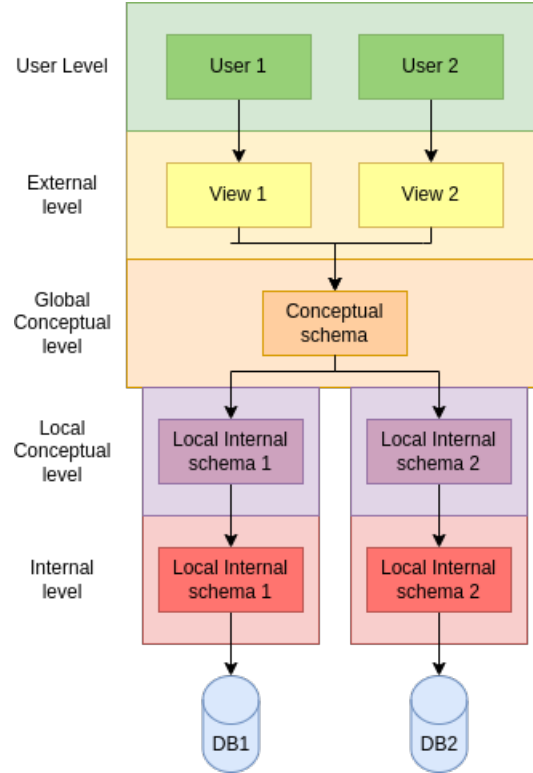
18

Figure 4: Extended ANSI/SPARC architecture with distribution.

Once these steps are done, the **execution manager** launches the different operators in the access plan in order, building up the results appropriately.

The **scheduler** deals with the problem of keeping the databases in a consistent state, even when concurrent accesses occur, preserving isolation (I from ACID).

The **recovery manager** is responsible for preserving the consistency (C), atomicity (A) and durability (D) properties.

The **buffer manager** is responsible for bringing data to main memory from disk, and vice-versa, communicating with the operating system.

This architecture is not sufficient to deal with distributed data. The functional architecture of a distributed DBMS (DDBMS) is depicted in Figure 6. As we can see, there are now two stages:

1. Modules cooperate at the global level, transforming the data flow and mapping it to the lower layers, dealing with a single view of the database and the distribution transparency:

   (a) The **global query manager** contains the view manager, security manager, constraint checker and query ooptimizer, which behave as in the centralized case, except for the optimizer, which now considers data location and consults the global schema to determine which node does what.

   (b) The **global execution manager** inserts communication primitives in the execution plan and coordinates the execution of the pieces of the query in the different components to build up the final results from all the query pieces executed distributedly.

   (c) The **global scheduler** receives the global execution plan and distributes trasks between the available sites, guaranteeing isolation between different users.

2. Modules cooperate at the local level, with a very similar behavior to that of the centralized DBMS.

3.7   Enumerate the eight main features of Cloud Databases
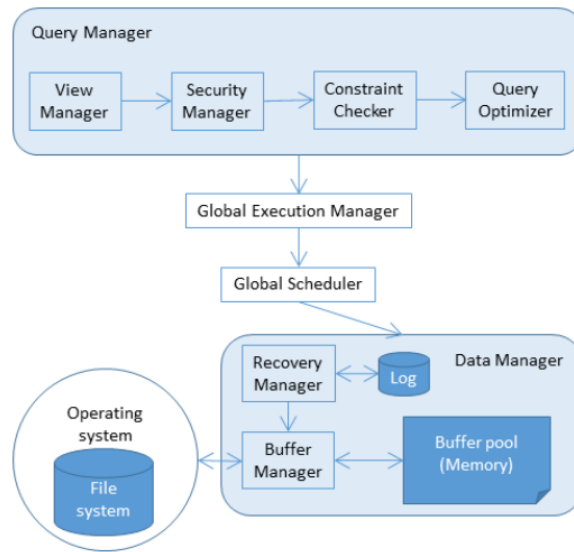
- Ability to scale horizontally.

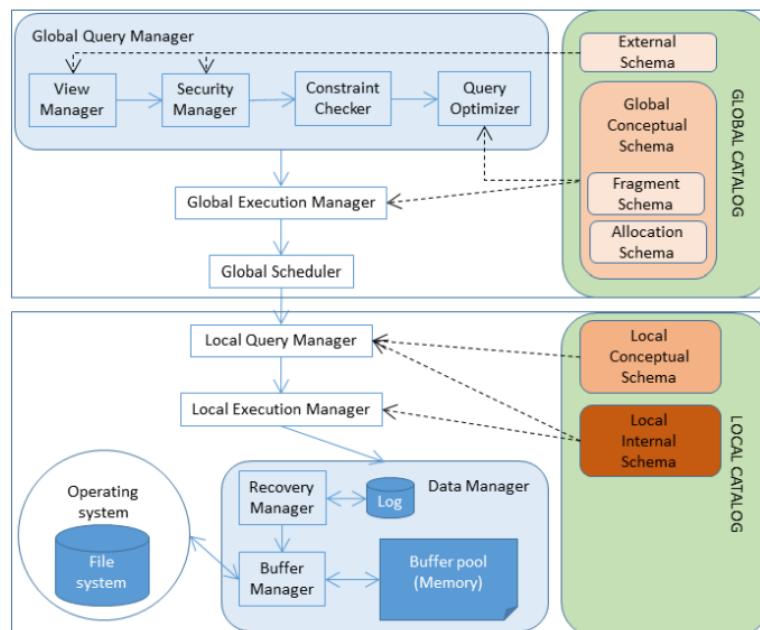Figure 5: Functional architecture of a centralized DBMS.

Figure 6: Functional architecture of a DDBMS.

- Efficient fragmentation techniques.

- Use as efficiently as possible distributed memory and indexing mechanisms to parallelize execution. Spped up relies on massive replication and parallelism (which in turn improve reliability and availability).

- Cloud Databases relax the strong consistency asked by ACID transactions and define the weaker concept of eventual consistency.

- A simplistic call level interface or protocol is provided to manage data, which is easy to learn and use, but puts the optimization burden on the side of the developers. This also compromises some transparency. The schemaless nature of these systems complicates even more the creation of a declarative query language like SQL.

- The setting up of hardware and software must be quick and cheap.

- The concept of **multi-tenancy** appears: the same hardware/software is shared by many tenants. This requires mechanisms to manage the sharing and actually benefitting from it.

- Rigid pre-defined schemas are not appropriate for these databases. Instead, there is a need towards gaining flexibility.

3.8 Explain the difficulties of Cloud Database providers to have multiple tenants

The difficulty can be summarized as the need to deal with the potential high number of tenants, and the unpredictability of their workloads' characteristics. Popularity of tenants can change very rapidly, fact that impacts the Cloud services hosting their products. Also, the activities that they perform can change.

Thus, the provider has to implement mechanisms to be able to deal with this variety and variability in the workloads.

Also, the system should tolerate failures and offer self-healing mechanisms, if possible.

Finally, the software should easily allow to scale out to guarantee the required latencies. Adding or upgrading new machines should happen progressively, so that service suspension is not necessary at all.

3.9 Enumerate the four main problems tenants/users need to tackle in Cloud Databases

- **Data design**: provides the means to decide on how to fragment the data, where to place each fragment, and how many times they will be stored (replication).

- **Catalog management**: requires the same considerations as the design of the database regarding fragmentation, locality and replication, but with regard to metadata instead of data. The difference in this case is that some of the decisions are already made on designing the tool and few degrees of freedom are left for administrators and developers.

- **Transaction management**: it is specially hard and expensive in distributed environments. Distributed recovery and concurrency control mechanisms exist, but there is a need to find a trade-off between the security they guarantee and the performance impact they have.

  Specially relevant in this case is the management of replicas, which are expensive to update, but reduce query latency and improve availability.

- **Query processing**: it must be as efficient as possible. Parallelism should benefit from data distribution, without incurring in much communication overhead, which can be reduced by replicating data.

3.10 Distinguish the cost of sequential and random access

3.11 Explain the difference between the cost of sequential and random access

3.12 Distinguish vertical and horizontal fragmentation

Data fragmentation deals with the problem of breaking datasets into smaller pieces, decreasing the working unit in the distributed system. It has been useful to reflect the fact that applications and users might be

interested in accessing different subsets of the data. Different subsets are naturally needed at different nodes and it makes sense to allocate fragments where they are more likely to be needed for use. This is **data locality**.

There are two main fragmentation approaches:

- **Horizontal fragmentation**: a selection predicate is used to create different fragments and, according to an attribute value, place each row in the corresponding fragment.

  A distributed system benefits from horizontal fragmentation when it needs to mirror geographically distributed data to facilitate recovery and parallelism, to reduce the depth of indexes and to reduce contention.

  Fragmentation can go from one extreme (no fragmentation) to the other (placing each row in a different fragment). We need to know which predicates are of interest in our database. As a general rule: *the 20% most active users produce 80% of the total accesses*. We should focus on these users to determine which predicates to consider in our analysis.

  Finally, we need to guarantee the correctness:

    - Completeness: the fragmentation predicates must guarantee every row is assigned to, at least, one fragment.
    - Disjointness: the fragmentation preficates must be mutually exclusive (**minimality property**).
    - Reconstruction: the union of all the fragments must constitute the original dataset.

  We have only considered single relations for this analysis, but it is also possible to consider related datasets and fragment them together, this is called **derived horizontal fragmentation**. Let R, S be two relations such that R possess a foreign key to S and are related by means of a relationship r. In this case, S is the **owner** and R is the **member**. Suppose also that S is fragmented in n fragments $S_i, i = 1, ..., n$, and we want to fragment R regarding S using the relationship r. The derived horizontal fragmentation is defined as

  $$R_i = R \ltimes S_i, \ i = 1, ..., n,$$

  where $\ltimes$ is the left-semijoin[1] and the joining attributes are those in r.

  If R and S are related by more than one relationship, we should apply the following criteria to decide which one to use:

    - The fragmentation more used by users/applications.
    - The fragmentation that maximizes the parallel execution of the queries.

  In order to consider a derived horizontal fragmentaiton to be complete and disjoint, two additional constraints must hold on top of those stated before:

    - Completeness: the relationship used to semijoin both datasets must enforce the referential integrity constraint.
    - Disjointness: the join attribute must be the owner's key.

- **Vertical fragmentation**: partitions the datasets in smaller subsets by projecting some attributes in each fragment.

  Vertical fragmentation has been traditionally overlooked in practice, because it worsened insertions and update timems of transactional systems in many times. However, with the arrival of read-only workloads, this kind of fragmentation arose as a powerful alternative to decrease the number of attributes to be read from a dataset.

  In general, it improves the ratio of useful data read and it also reduces contention and facilitates recovery and parallelism.

  As disadvantages, note that it increases the number of indexes, worsens update and insertion time and increases the space used by data, because the primery key is replicated at each fragment.

  Deciding how to group attributes is not obvious at all. The information required is:

    - Data characteristics: set of attributes and value distribution for all attributes.

---

[1] $R \ltimes S$ pairs those tuples in R for which there is at least one tuple in S with matching joining key.

– Workload: frequency of each query, access plan and estimated cost of each query and selectivity of each predicate.

A good heuristic is the following:

1. Determine primary partitions (subsets of attributes always accessed together).
2. Generate a disjoint and covering combination of primary partitions, which would potentially be stored together.
3. Evaluate the cost of all combinations generated in the previous phase.

3.13   Recognize the complexity and benefits of data allocation

Once the data is fragmented, we must decide where to place each segment, trying to optimize some criteria:

- **Minimal cost**: function resulting of computing the cost of storing each fragment F_i at a certain node N_i, the cost of querying F_i at N_i and the cost of updating each fragment F_i at all places where it is replicated, and the cost od communication.

- **Maximal performance**: the aim is to minimize the response time or maximize the overall throughput.

This problem is NP-hard and the optimal solution depends on many factors.

In a dynamic environment the workload and access patterns may change and all these statistics should always be available in order to find the optimal solution. Thus, the problem is simplified with certain assumptions and simplified cost models are built so that any optimization algorithm can be adopted to approximate the optimal solution.

There are several benefits to data allocation, including:

- Improved performance: By distributing the data across multiple nodes, the workload can be distributed among the nodes, reducing the load on any single node and improving overall performance.

- Increased availability: With data replicated across multiple nodes, the failure of any single node does not result in a loss of data or loss of access to the data.

- Scalability: Distributed data allocation allows for scaling the system by adding more nodes to the system, as needed.

- Reduced network traffic: By keeping data local to the nodes where it is most frequently accessed, data allocation can reduce the amount of network traffic needed to access the data.

- Better resource utilization: Data allocation can help to balance the use of resources across the nodes in the system, avoiding overloading some nodes while underutilizing others.

3.14   Explain the benefits of replication

Data replication refers to the process of making and maintaining multiple copies of data across multiple nodes in a distributed database system. There are several benefits to data replication, including:

- Improved availability: By replicating data across multiple nodes, the system can continue to function even if one or more nodes fail or become unavailable, ensuring the availability of the data.

- Increased fault tolerance: Data replication can help to ensure that data remains available even in the event of a hardware or software failure, improving the overall fault tolerance of the system.

- Faster data access: With multiple copies of data available across multiple nodes, data can be accessed more quickly by users and applications, improving overall system performance.

- Improved load balancing: Replicating data across multiple nodes can help to balance the workload on each node, improving overall system performance and efficiency.

- Enhanced data locality: Replication can also help to improve data locality by ensuring that frequently accessed data is available on the same node, reducing the need to access data over the network.

3.15   Discuss the alternatives of a distributed catalog

The same design problems and criteria can be applied to the catalog, but now we are storing metadata. This requires two important considerations:

1. Metadata is much smalles than data, which makes it easier to manage.

2. Optimizing performance is much more critical, since accessing this metadata is a requirement for any operation in the system.

Many decisions are already made by the architects of the system, and only few options can be parameterized on instantiaitng it.

- **Global metadata**: are allocated in the coordinator node.

- **Local metadata**: are distributed in the different nodes.

A typical choice we can make in many NOSQL systems is having a secondary copy of the coordinator (**mirroring**) that takes control in case of failure. Of course, this redundancy consume some resources.

3.16   Decide when a fragmentation strategy is correct

References

[1] Alberto Abelló and Sergi Nadal. Big data management. Lecture Notes.