Big Data Management

Alberto Abelló Victor Herrero Moditha Hewasinghage Rana Faisal Munir
Sergi Nadal Oscar Romero

Database Technologies and Information Management (DTIM) group
Universitat Politècnica de Catalunya (BarcelonaTech), Barcelona
February 15, 2021

Foreword

Data drives the world. According to the 2030 Agenda for Sustainable Development, "*Quality, accessible, timely and reliable disaggregated data will be needed ...*" to drive actions, and monitor and verify achievements. Market analysis firm McKinsey states "*Data is now a critical corporate asset. It comes from the web, billions of phones, sensors, payment systems, cameras, and a huge array of other sources–and its value is tied to its ultimate use. While data itself will become increasingly commoditized, value is likely to accrue to the owners of scarce data, to players that aggregate data in unique ways, and especially to providers of valuable analytics*." The European Political Strategy Centre (EPSC) confirms "*Data is rapidly becoming the lifeblood of the global economy. It represents a key new type of economic asset. Those that know how to use it have a decisive competitive advantage in this interconnected world, through raising performance, offering more user-centric products and services, fostering innovation—often leaving decades-old competitors behind. [. . .] Data analytics will soon be indispensable to any economic activity and decision-making process, both public and private*." While attention to the importance of data is not new, the predicted economic value that can be extracted from the availability of these data remains as of yet largely unrealized. According to McKinsey, "*Most companies are capturing only a fraction of the potential value from data and analytics. [. . .] manufacturing, the public sector, and health care have captured less than 30 percent of the potential value we highlighted five years ago.*" While there are multiple factors responsible for the non-realisation of this potential, a crucial factor is that unlocking value from raw data is hard: before any newly acquired data becomes useful, it must be preprocessed, integrated with existing data, cleaned from errors, appropriately stored, and prepared for analysis. Further, usage is no longer restricted to simple querying or data mining, it increasingly requires exploration, recommendation, and explanation along with special treatment of time and location. Therefore, if quality is defined by "fitness for use", the quality (and thus, the value) of incoming data is very low at its inception and a great deal of management and processing effort is required to increase this.

A typical data value creation chain encompasses multiple disciplines and people with different roles, of which Data Science and Data Engineering are two prominent examples. As defined in [Dha13], **Data Science (DS)** is the scientific "*interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from data in various forms, both structured and unstructured.*" In other words, data scientists focus on extracting meaning and insight from data through analytics. They are supported in their activities by **Data Engineering** (DE). Data engineers "*design and build the data ecosystem that is essential to analytics. Data engineers are responsible for the databases, data pipelines, and data services that are prerequisites to data analysis and data science*." In setting up these pipelines and functionalities that conform the ecosystem, data engineers are often faced with the challenges posed by extreme characteristics of **Big Data (BD)**, defined by the Oxford English Dictionary as "*data of a very large size, typically to the extent that its manipulation and management present significant logistical challenges*." Addressing these challenges demanded new technological solutions for **data management** ("*architectures, policies, practices and procedures that properly manage the full data lifecycle needs of an organisation*".

Marie Skłodowska-Curie ITN proposal DEDS[1], 2020

[1] `https://deds.ulb.ac.be`

Contents

Chapter 1

Introduction

Any data-driven organization can be divided in three parts or subsystems:

Production system devoted to fulfill its purpose.

Decision system that defines how the goal of the company has to be achieved and defines the behaviour of the other subsystems.

Information system that manages the information generated and used by the others.

This division of organizations is generic (i.e., applies to both profit and non-profit organizations) and not new, despite it was only in the second half of 20^{th} century that computers were introduced in the management of information. In the past, the management of information was mainly manual (i.e., paper files) generated and used in the production system. It was later that such information was also used to make decisions, and more recently also information external to the company started to be used. Data is the representation of that information.

1.1 Data Driven decision making

It was in the 90's those organizations started to systematically use data in decision making. As previously said, data had always been used in any organization, but it was just then beginning to be pervasive in all the processes and cheaper to store and manage. Thus, the concept of Business Intelligence (BI) appeared to talk about dashboards representing the status and evolution of companies. These basic dashboards used to show bar, line or pie charts with the key performance indicators of the business.

Data to create those dashboards came mainly from the different applications being used by the production systems. Nevertheless, to avoid interfering those applications, they were Extracted, Transformed and then Loaded (ETL) into dedicated databases known as Data Warehouses (DW). Different kinds of DW were interconnected by ETL flows in a complex Corporate Information Factory (see [CIF02]) integrating data coming from multiple sources in different formats (e.g., Relational, XML, HTML). To facilitate the access to those data once integrated, they were shaped following the multidimensional model into data cubes to be exploited by OLAP (online-analytical processing tools, see [AR18]). These allowed three access patterns: (1) static generation of reports, (2) dynamic (dis)aggregation and navigation by means of OLAP operations, and (3) inference of hidden patterns or trends with data mining or Machine Learning (ML) tools. The latter was typically performed out of the DW, but today the current trend in RDBMS is to include native implementation of advanced ML functionalities.

Figure 1.1 sketches the whole process. At the bottom, we can find the different applications serving the production system in its day by day operations. Then, as part also of the data management tasks of the company, multiple ETL flows extract their data to load them into the DW (both together comprise what we call data warehousing). The three kinds of analytical tools would be plugged to the DW for decision making purposes. Their results would in the end be used by the executives to change or reinforce the business strategy, which would obviously impact the data being collected by operational applications.

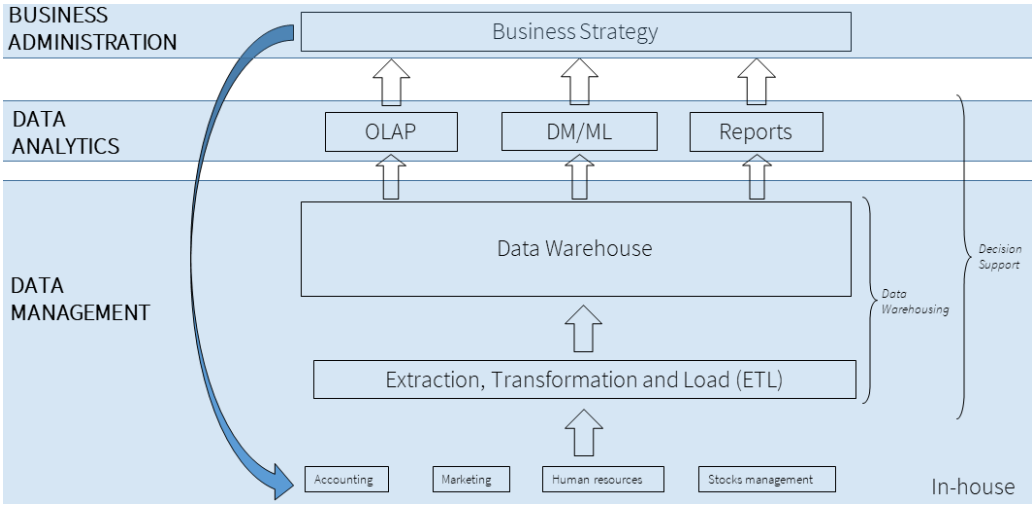


Figure 1.1: Business Intelligence Cycle

1.2 Big Data

Big Data is a natural evolution of BI, and inherits its ultimate goal of transforming raw data into valuable knowledge. Nevertheless, traditional BI architectures, whose de-facto architectural standard is the DW, cannot be reused in Big Data settings. It is necessary to shift from the well known and comfortable DW environment to a wilder and open world where we consider sources of information external to the company and its transactional systems. This shift is allowed by the development in the computational capacity (see Section 1.3), but actually provoked by the opportunity offered by the presence of humungous amounts of data (mainly coming from social networks and sensors). In the 20^{th} century, the WWW became less passive and more pervasive with the appearance of new devices and the offer of new features and use cases. It shifted from static hand-crafted contents provided by few gurus, to dynamic contents easily generated by anybody from anywhere in the World, specially through social networks. In parallel to this, industry improved also automation and digitalization/sensorization allowing to monitor basically any production or service process, giving rise to the concept of Internet of Things (IoT), whose sensors generate a never ending continuous flow of information. Indeed, the so-popular characterization of Big Data in terms of the three "V's (Volume, Velocity and Variety)", refers to the inability of DW architectures, which typically rely on Relational databases, to deal and adapt to such large, rapidly arriving and heterogeneous amounts of data.

Figure 1.2 highlights the differences between BI and Big Data. Firstly, we can appreciate a new border between the information system and its sources, which are now somehow out of the data management block. This does not mean all sources are external to the company, but some are, and this is one of the causes of the change in paradigm. The lack of control this entails in the sources requires mode flexibility in the data flows and storage, provoking a technological and name change, diving rise to dynamic data intensive flows and data lakes (instead of more rigid ETL and DW). In the analytics, because of the same reason, there is also a change in technology and terminology. Small and Big Analytics concepts refer to performing traditional OLAP/Query&Reporting to gain quick insight into the datasets by means of descriptive analytics (i.e., Small Analytics) and Data Mining/Machine Learning to enable predictive analytics (i.e., Big Analytics) on Big Data systems, respectively.

Next, we describe the technical characteristics that make up Big Data, and the challenges that these entail both from a data perspective and from an analysis perspective.

1.2.1 Big Data, from a data management perspective

As we can see in the benchmarks defined for the purpose of testing the systems dealing with it (see [GRH+13]), the difference between having Big Data or not comes from incorporating huge external not necessarily structured sources to those already existing in our organization. These data can come from partner organizations (typically structured), or from completely independent sources like social networks (completely unstructured and in the form of natural language). There is, indeed, a middle ground where organizations incorporate to their analysis semi-

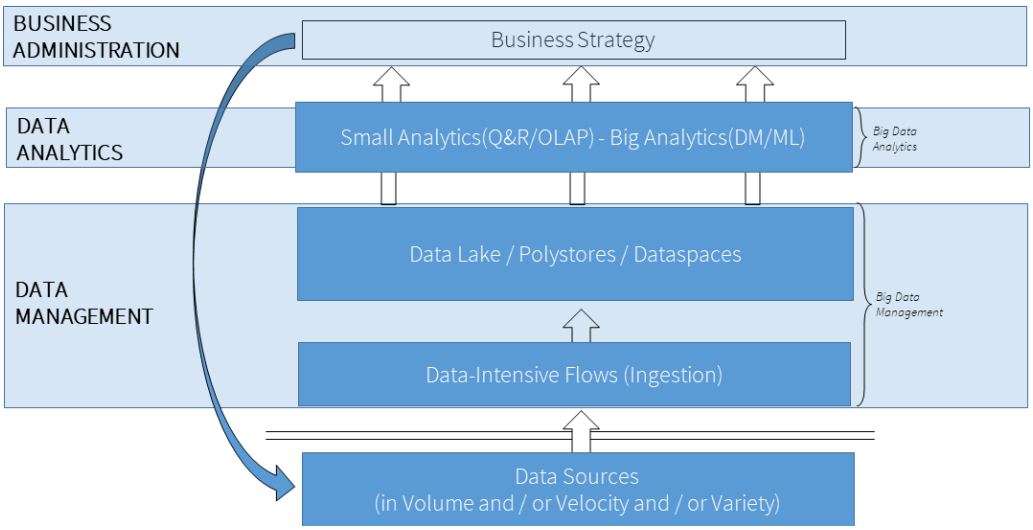


Figure 1.2: Big Data Cycle

structured data (like logs).

As a consequence of incorporating those sources to decision making, new requirements and challenges appear to characterize Big Data. The most popular characterization for Big Data systems is in terms of 5 V's[1]. We present and detail each of these dimensions below.

Volume refers to the large amount of digital information produced and stored in these systems, nowadays shifting from terabytes to petabytes. The most widespread solution for Volume is data distribution and parallel processing, typically using Cloud-based technologies.

Velocity refers to the pace at which data are generated, ingested (i.e., dealt with the arrival of), and processed, usually in the range of milliseconds to seconds. This gave rise to the concept of data stream and creates two main challenges. First, data stream ingestion, which relies on a sliding window and buffering model to smooth arrival irregularities. Second, data stream processing, which relies on linear or sublinear algorithms to provide near real-time analysis.

Variety deals with the heterogeneity of data formats, schema and how to deal with their integration, paying special attention to semi-structured and unstructured external data (e.g., text from social networks, JSON/XML-formatted scrapped data, Internet of Things sensors, etc.). In March 2016, the MIT Sloan Management Review Group identified the *data variety challenge* as the most crucial challenge in data-driven organizations. Addressing the challenge requires providing on-demand integration of an heterogeneous and evolving set of data sources. Aligned with it, the novel concept of Data Lake has emerged, a massive repository of data in its original format. Unlike the DW that follows a *schema on-write* approach, the Data Lake proposes to store data as they are produced without any preprocessing until it is clear how they are going to be analyzed, following the *load-first model-later* principle. The rationale behind a Data Lake is to store raw data and let the data analyst decide how to cook them.

Variability is concerned with the evolving nature of ingested data, and how the system copes with such changes for data integration and exchange. In the Relational model, mechanisms to handle evolution of *intension* (i.e., schema-based), and *extension* (i.e., instance-based) are provided. However, achieving so in Big Data systems entails an additional challenge due to the schemaless nature of NOSQL databases.

Veracity has a tight connection with data quality, achieved by means of data governance protocols. Data governance concerns the set of processes and decisions to be made in order to provide an effective management of the data assets. This is usually achieved by means of best practices, where systematic data quality control and cleaning is enforced.

[1] Originally, Gartner included only the first three.

1.2.2 Big Data, from a data analysis perspective

Another popular characterization for Big Data systems is in terms of the type of analysis to be performed. We distinguish between three types: descriptive, predictive and prescriptive.

Descriptive analysis uses basic statistics like min, max, stdev, avg, etc. to describe the data. In a DW environment, this is typically done interactively by means of Online Analytical Processing (OLAP) tools, which allow to dynamically modifying the analysis point of view (moving up and down in aggregation hierarchies) to facilitate the understanding and gain knowledge about the stored data. The main idea is to provide users navigation mechanisms in a graphical interface so that they can easily generate queries on the fly.

Predictive analysis encompasses a set of statistical techniques such as data mining, predictive modeling or machine learning, which analyzes historical facts with the goal of making predictions about future events. Predictive analysis is commonly run in specialized tools (e.g., R or SAS), which provide an extensive toolbox of algorithms and utilities. This, however, brings several challenges for data management, as data are dumped from DWs into files, which are later feed as input to these tools. This, clearly, is highly inefficient, as the database structures are not leveraged. Providing methods and tools for in-database predictive analysis is nowadays a hot research topic (see [Olt20]).

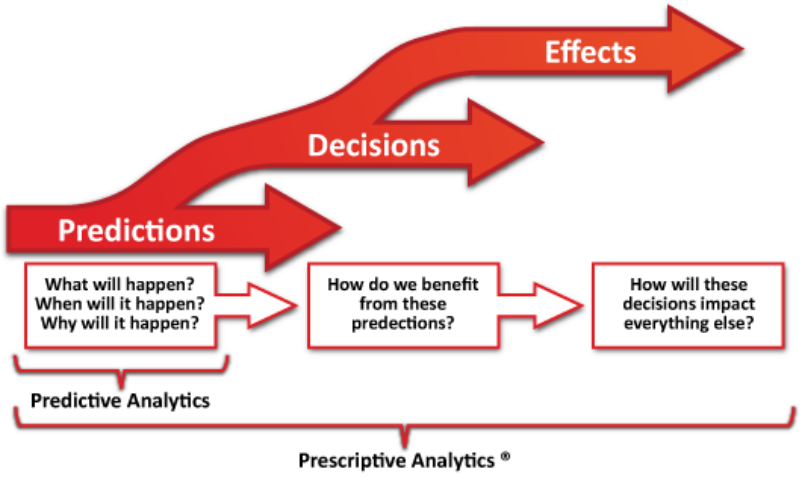


Figure 1.3: The three phases of prescriptive analytics, by Modaniel, CC BY-SA 4.0, via Wikimedia Commons

Predictive analysis takes as input the predictions of previous analysis to suggest actions, decisions and present the possible implications of each of the them. Figure 1.3 depicts the phases that make up prescriptive analysis. This kind of analysis is nowadays still in its inception, being recommender systems those that implement this vision (but always requiring the end-user's final validation).

1.3 Cloud Computing

In the beginning of electricity, every industry had own a generator if they wanted to use it. Eventually, a public network was created and big generators (e.g., dams) started serving in to everybody in a more efficient and clean way. Today, still some organizations (e.g., hospitals) have electricity generators, but most of them simply connect to the public network. We can consider computing just another commodity and think of it simply as one more service that someone can provide for us.

According to the American National Institute of Standards and Technology (NIST), "*Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" We can think of it like a new business model to

commercialise computing power like we commercialise electricity. In the same way that not every home has a dam, not every company needs to own and manage its data center.

This introduces some novelty in that it eliminates any upfront investment (you do not need to buy an expensive piece of hardware before start working), but overall generates the illusion of infinite resources at your disposal. Moreover, you pay exactly for what you use (if you do not use the computing power, you do not pay for it), and providers guarantee service levels that can only be achieved thanks to the economy of scale in software development and management.

Indeed, the main benefit of the Cloud comes from such economy of scale that allows to reduce costs and be more efficient. A machine hosted in-house is most of the time underused, because usually organizations do not require it being 100% operational all the time. However, in a global world, if that machine is made available as a service to millions of customers, there will always be someone somewhere that requires its computational capacity. On the other hand, service customers can flexibly adapt their costs to their real needs at any time. Moreover, the IT department does not need to take care of the hardware management, maintenance or upgrading.

This paradigm originated in hardware, but can be applied in the same way at any point in the software stack. Most typical levels are:

Infrastructure as a Service (IaaS) virtualizes the hardware and operating system.

Platform as a Service (PaaS) offers the remote execution of basic packages, or modules (e.g., database, authentication) required as part of bigger and complex applications. These are seamlessly integrated like any other library.

Software as a Service (SaaS) offers some software packages ready to be used by final users, typically through the graphical interface of a web browser (e.g., Google docs, Dropbox).

Business as a Service (BaaS) creates small stand-alone pieces of software that can be easily combined to create business flows in interaction with others pieces from potentially other service providers (e.g., Paypal, Amadeus).

1.4 Big Data Management Systems

With the new requirements in both the data and the infrastructure came new tools to deal with them. Thus, in the 21^{st} century appeared the NOSQL movement, standing for Not Only SQL, which showed the need of breaking with the traditional RDBMS systems. However, despite the catchy name, the problem had nothing to do with SQL but with simplifying unnecessary features that compromised performance, allowing scalability on commodity hardware characteristic of the Cloud, increasing availability and reliability, and at the same time lowering management complexity.

Thus, it is clear that first of all, these systems must definitely be distributed. However, we still have some open choices both from the perspective the database model they offer (with implications in the query interface) and from the internal implementation they use (see [TADP21] for a brief discussion of alternatives).

Regarding the database model, they relax the classical tabular (a.k.a. Relational) structure, by allowing semi-structured documents, wide-columns or simple key-values, but also more generic and semantically rich graphs. Regarding the implementation, they can rely on simple in-memory structures, sequential disk accesses, hash-based maps (e.g., consistent hash or linear hash), or some kind of tree structures (e.g., B-tree or LSM-tree). Both are equally relevant, since one affects project development and maintenance, but the other (which is hidden and often ignored) heavily impacts functionalities and performance. Consequenly, we will have to analyze each tool from these perspectives. Chapter 2 explains the foundations for the former and Chapter 3 for the latter.

Chapter 2

Big Data Design

It is widely accepted today that Relational databases are not appropriate in highly distributed architectures of commodity hardware, that need to handle poorly structured heterogeneous data. Thus, the change in the data model and the new analytical needs beyond OLAP forced the rethinking of methods and models to design and manage these newborn repositories. This has brought the blooming of alternative systems with the purpose of mitigating such problem, specially in the presence of analytical workloads. Many of the required techniques and theoretical concepts in these systems have been present for a long time (see [ÖV20]), but it is only the confluence of both the opportunity and the means (a.k.a. new data sources and Cloud computing) at the beginning of 21^{st} century that makes them a reality. Such an important change in the technologies brings also an associated challenge in the database design, where some performance factors that had not been considered important regain focus, mainly because of data distribution and lack of data structure. Thus, the existing tool set (e.g., B-tree) and method (e.g., Relational normalization theory) prove to be useless in this new context, and need to evolve to cope with Variety and consider new access patterns.

2.1 Motivation

Traditionally, databases were used in the operation of companies, where registering data in the form of transactions is necessary. This created the solid Relational theory in the 70's that has successfully lasted until today. Nevertheless, there are other non-clerical uses where the tabular pattern is not suitable (e.g., OLAP, scientific data, semantic web, agile development, real-time events, etc.). To that end, new data models such as graphs, key-values or semi-structured documents provide the needed flexibility to accommodate new requirements and overcome the traditional rigidity and staticity of Relational databases.

Relational stores atomic values in the intersection between rows and columns.

OLAP analysis is based on the multidimensional model that clearly separates what is the focus of analysis (a.k.a. fact and its measures) and its descriptors (a.k.a. dimensions).

Key-Value manage data in hash tables or dictionaries where you can put some values (i.e., sets of bytes) that then can be retrieved given the associated key.

Wide-column also known as Column-family, extend Key-Value by allowing vertical partitioning of values.

Graph highlight the relevance of relationships between data and represent them in terms of nodes and edges generating free topologies.

Documents are graphs restricted to tree patterns, which facilitates packing the storage of multiple nodes hierarchically.

Obviously, all these database models did not appear all the sudden, but rather slowly evolved. Figure 2.1 illustrates that evolutions and their theoretical foundations.

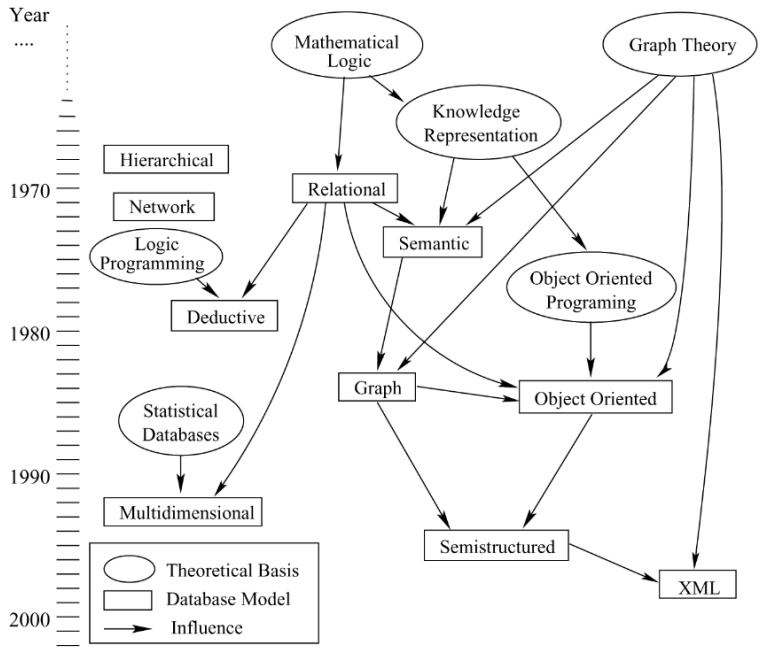


Figure 2.1: Evolution of database models from [AG08]

2.2 Schema definition

It is clear that the proliferation of database models and storage mechanisms hinders the application development task. Ideally, we would like that our program instructions are completely independent of how data is physically represented and store. However, this is simply impossible.

2.2.1 ANSI/SPARC architecture

In the 60's, before the Relational model, several alternatives co-existed to store data (e.g., hierarchical, network, etc.). Then, the big achievement of the Relational model was to provide a higher level of abstraction, that made data management independent of how data were physically stored.

Figure 2.2: ANSI-SPARC architecture

By that time, ANSI[1] created SPARC[2] Study Group on DataBase Management Systems. The main contribution of that group was to propose a DBMS architecture (see [Jar77]). This architecture, sketched in Figure 2.2, defined three different levels for DBMSs to implement. At the RHS, we had the physical one corresponding to files and data structures like indexes, partitions, etc. To its left laid the table according to E. Codd's Relational abstraction. Finally, at LHS, different views could be defined to provide semantic relativism (i.e., each user can see the data from her viewpoint, namely terminology, format, units, etc.).

ANSI-SPARC architecture provided, on the one hand, logical independence (i.e., changes in the tables should not affect the views from users perspective), and, on the other hand, physical independence (i.e., changes in the files or data structures should not affect the way to access data). This Relational feature was really important,

[1] American National Standards Institute
[2] Standards Planning And Requirements Committee

firstly because it made a difference with regard to predecessors, but also because, at the time, it was still not set the best way to store or index Relational data. To store a table, you can clearly consider two options: row-wise and column-wise (see [CK85]).

From a database design perspective, in practice, there are also three phases: Conceptual, logical and physical. Physical design corresponds to the physical schema in the ANSI-SPARC architecture, while the logical one encompasses both conceptual as well as external schemas, following the Relational model. Nevertheless, from an engineering perspective, it is much easier to raise the abstraction level and use some semantic data model closer to human thinking. The choice for such semantic model has traditionally been Entity/Relationship (E/R) or lately also its descendant UML[3]. The advantage of this approach is that fairly mechanical transformations exist to translate from both into the Relational model (see [GMUW09]). This, together with normalization theory (see [AHV95]), has provided solid foundations to validate the correctness of Relational database designs.

Back to the past in the pre-Relational world, we are now again at a crossroad where correctness of database design is not clear anymore. Big Data analytics demands high efficiency, specially to achieve Velocity in the presence of high data Volume. The way to answer such demand has been moving away from generic RDBMSs and deploy specialized architectures (see [Sto08]). Thus, every kind of dataset requires different features in the management system, and even different data models (i.e., Sequential files, Key-Value stores, Document stores, Wide-column stores, etc.) and query languages (usually just APIs), which are neither declarative nor standardized, resulting in loss of physical independence.

2.2.2 New requirements

The only problem of Big Data is not Volume, but also Velocity. This can be seen from two different perspectives. Clearly, it refers to response time, but we should not forget the time to store the data as it arrives. Arrival rate can be high to the point of having streams (e.g., IoT and Smart City sensors). Firstly, true loading (including parsing and formatting) is simplified into just ingesting (without digesting) data, following a "data-first model-later" approach. Secondly, the traditional ETL paradigm in DW, needs to become just EL (without "T"), with really light-weight transformations and no integration at all. Thus, it is now the responsibility of analysts to perform ad-hoc integration and cleaning of data. This can clearly impact on data quality, which in the form of Veracity is also recognized by some authors as another "V" for Big Data.

Multidimensional modeling (see [AR18]), the de-facto standard for DW, is a simple yet powerful metaphor that focuses on subjects of analysis and their facets, which is implemented with a Star-join Relational schema. However, the Star-join schema (see [GMUW09]) is not appropriate for flexible Big Data settings since not only the subject, but also the potential dimensions of analysis are fixed at design time. Furthermore, adding new dimensional or factual data is a costly operation in the DW, since it is typically implemented with Relational technology.

Thus, the design paradigm has to evolve accordingly, and Star-join Schemas are not appropriate anymore for DW. This trend reaches the extreme in the form of Data Lake[4] (DL), which is a repository (usually distributed and heterogeneous) where raw data is stored in waiting for an analytical purpose being defined. Of course, the risk of just throwing data into a DL without keeping track of semantics and structure is that it becomes a swamp. In order to avoid it, we need clear Data Governance rules and also store the associated metadata. Thus, to start with, we should pay attention to how the repository should be designed and analytical needs modeled.

The required diversity in the physical schema to deal with ingestion problems raises several challenging design issues. Some of them where already solved by RDBMSs, others just gain focus due to the need of scaling out, and others (like normalization theory) just need to be reviewed simply because assumptions (i.e., redundancy avoidance) are not valid anymore.

Store choice: Some Relational systems already offer more than one storage engine (e.g., InnoDB and MyISAM in MySQL). However, diversity in NOSQL is much higher and continuously changing, and the choice can heavily impact performance. We would expect they eventually converge and stabilize to make the selection much easier, but this is not the case, yet.

Key design: Also in RDBMSs, deciding the key of each table is important, but already solved. However, because not exactly corresponding to the Relational definition and being used to distribute (and in some cases

[3] Unified Modelling Language

[4] The term was coined by James Dixon (Pentaho) in his blog (`http://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes`).

physically sort) data, its choice has consequences in Big Data locality. Because of the utmost importance of performance, some physical characteristics like the selectivity factor and attributes considered in query predicates have to gain relevance in the parameters of the database design, see [RHAF15]. Besides, balancing workload among machines in the cluster, where data is located can impact the cost of many algorithms, so we need to take into account in the database design which and how many data each functionality is going to use.

Denormalization: Write-intensive RDBMS (in operational environments) use normalization to avoid redundancy and therefore insert, update and delete anomalies. Oppositely, read-intensive systems (in decision support) use denormalization in order to avoid joins and improve performance. Indeed, the plain structure of $1NF$ Relations (according to normalization theory, see [AHV95]) has simplified the design for many years. Now, the popularity of JSON documents not only allows, but also promotes NF^2, raising the question of which is the best nested structure for each document. Collapsing one-to-many relationships in one document can reduce I/O if they are usually accessed together, but can also reduce the hit ratio in the cache if they are not. Conversely, replicating data in many-to-one relationships can save some joins by introducing redundancy, with the corresponding extra storage and update overhead.

Integrity management: Even in RDBMS, referential integrity and constraint checking is often disabled. In NOSQL systems, enforcing them is simply not possible, because they do not provide such functionalities for the sake of performance gain.

Horizontal partitioning: This is a question one has to answer in RDBMSs too. Nevertheless, it is more complex and relevant in Big Data systems where partitions are distributed in a number of machines that has also to be decided. The more we distribute the data, the more we can parallelize. However, we firstly incur in communication costs, and also economical costs because of using more hardware.

Vertical partitioning: Again, this is not new or inherent to Big Data, but the nature of analytical queries and the importance of performance underlines its relevance. In RDBMSs, the choice was disguised in the form of nested tables in Oracle or inheritance in PostgreSQL. Although, pure column stores appeared like Vertica or MonetDB[5], or even others like SAP HANA[6] offer both possibilities, the approach to chose between row or columnar storage is still purely heuristic.

2.2.3 Co-Relational Model

Big Data has recently gained popularity and has strongly questioned Relational databases as universal storage systems, especially in the presence of analytical workloads. As a result, alternatives, commonly known as NOSQL (Not Only SQL) databases, are extensively used for Big Data. As the primary focus of NOSQL is on performance, their data are directly designed at the physical level, and consequently the resulting schema is tailored to the dataset and access patterns of the problem in hand. Moreover, a well known problem of RDBMS is the Impedance Mismatch (i.e., the overhead generated by transformations from internal structures to tables, and then into programming structures). Thus, these systems focus on new database models to reduce that. For example, in order to cope with such problem, some systems physically store JSON (JavaScript Object Notation) documents associated to the keys, which then directly map to the programming language in-memory structure. This entails a complete rethought of the internal structures as well as the means to couple data analytics on top of such systems.

In NOSQL tools, data is not stored into tables and foreign keys do not exist either. Instead, complex, nested structures are defined. Rather than as a negation, this should be seen as complementary to the Relational model (see [MB11]), where instead of having children pointing to parents, it is the parent who contains (or points to) the children.

Taking *Relational* stores as reference baseline, we are going to consider three other co-Relational models, namely *Key-Value*, *Wide-column*, and *document*. In general, Key-Value stores (see Chapter 5 for details) allow to dynamically and flexibly associate values to column names (i.e., there is not pre-defined schema or data types inside the value). Terminology adopted by these systems can confuse those who try to find a matching with Relational concepts. In this case, a table is just a set of Key-Value pairs (i.e., a hash table), and columns must be interpreted

[5] `http://www.monetdb.org`
[6] `https://www.sap.com/products/hana.html`

as simple tags that help partial retrieval of the value. Some of these stores extend also the concept of column with that of Column-Families, giving rise to Wide-column systems. Such grouping of columns directly translates into a vertical partition of the table, and entails the consequent loss of schema flexibility. However, specially for analytical purposes, this is really convenient if we can identify affinities between columns that are usually accessed together. This concept was introduced in Google BigTable and is implemented in Apache HBase, too. On the other hand, document stores (see Chapter 6 for details), instead of associating black-box values to the keys, associate semi-structured documents (i.e., JSON or XML) with some schema information, so that secondary indexes can be defined on them.

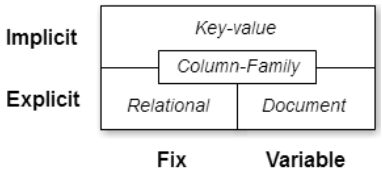


Figure 2.3: Database models depending on their schema properties

Figure 2.3 classifies their underlying structures with respect to the schema nature. A DBMS with implicit schema does not manage any information about the instance structure, which is a black-box for the system, and data must be parsed and interpreted only at the application level. We say that schemas are explicit if they are declared, which allows the database to automatically parse the instance data (roughly speaking, if the database can make any use of the instance structure, for example to generate and maintain secondary indexes). Explicit schemas can in turn be either fix or variable. In the former, all instances' data follow the same schema, which is globally declared once, while in the latter, instance data is individually embedded within its schema.

To exemplify those three models, we will use a toy example. Let us suppose the following information: "the city of Barcelona (BCN) has a population of 2,000,000 inhabitants and is located in Catalonia (CAT)". Introducing the internals of the Relational data model is not the purpose of this chapter, but it is worth to mention that its classification clearly falls into the explicit quadrant as it fully relies on a fix schema definition shared by all instances. This could indeed be captured in a single relation (*city*), with three attributes: *name*, *population* and *region*. Using SQL notation, it would look like: `city(name, population, region) VALUES ('BCN', '2,000,000', 'CAT')`. First an attribute *city* containing the value *BCN* and being the primary key, second an attribute *population* containing the value *2,000,000* and finally an attribute *region* containing the value *CAT*.

Key-Value stores have the simplest layout. Instance data is stored in tables and represented with a key (i.e., identifier), and a value (i.e., associated data). Neither the key nor the value are tied to a specific format, and behave as black-boxes for the system (the former is typically a string and the value a binary object). Our example could be represented as: `['BCN', '2,000,000;CAT']`. The application layer is responsible for parsing the value, and the region of Barcelona must be obtained in through splitting this instance value by semicolons and taking the second element. Query answering must then be achieved by means of ad-hoc procedures. Thus, no declarative query language and optimizer can be provided to access the data, and only simple actions (such as get and put by key) are offered via low-level APIs. Consequently, the application layer is responsible for interpreting each instance, and we consider the schema to be implicit, which is typically referred to as "schemaless". The only property this database model can assert is the unique identification of values through their corresponding keys, which is inherited from basic modeling principles, where it is stated that any instance must be unambiguously identified. An example of key-value store is Redis.[7]

Document stores are sets of potentially heterogeneous collections (i.e., namespace) containing different kinds of documents. A document is a just another Key-Value structure where the value is a semi-structured document (typically JSON or XML), that can in turn be seen as a set of entries in the form of (potentially nested) Key-Value pairs. Thus, the schema is explicit but variable, since the flexible XML or JSON structure is stored with the instance. In our example, the corresponding document would be `[id:'BCN', population:'2,000,000', region:'CAT']`. So, providing some structure to the value opens the door for higher-level query languages and optimization. Typically, document stores use a *query-by-example* approach, and given a pattern document (e.g.,

[7] `https://redis.io`

name = 'BCN' or salary > 5000), all documents fulfilling such pattern are retrieved. Thus, the application layer can benefit from the instance structure and query it by means of the key (retrieving the whole document) or by key plus attribute. The DBMS is then enhanced with advanced query capabilities and therefore it reliefs the application layer from parsing the instance. An example of document store is MongoDB.[8]

Wide-column stores are Key-Value stores that further structure the value into families, which contain groups of attributes (a.k.a. columns). Similar to documents, we could query and retrieve the whole instance, a family, or a specific attribute within a family. Families actually denote vertical fragments, and each is physically stored in a different disk file. From the schema point of view, families are static and defined at table creation time, whereas attributes can be dynamically specified at data insertion time (i.e., the attribute name is stored together with the instance data). Two different attributes may even have the same name as long as they belong to different families. Thus, the table schema is (i) explicitly declared (i.e., the families), static, and shared by all instances, but also (ii) explicit and variable within families, since attributes may vary among instances, and finally (iii) data types are implicit since the attribute value is stored in the form of Key-Values. Therefore, families and attributes give several degrees of freedom when designing the schema. Our example could result in a table with two families, namely *population* and *region*, and a constant attribute in each of them, named *value*, to store each attribute: `['BCN', population:{value:'2,000,000'}, region:{value:'CAT'}]`. Alternatively, we could use a single family *all* containing both attributes: `['BCN', all:{population:'2,000,000',region:'CAT'}]`. Similar to Key-Value stores, we may also use a single column inside the family: `['BCN', all:{value:'2,000,000;CAT'}]`. An example of Wide-column store is Apache HBase.[9]

Once reached this point, it is very important not to confuse Wide-column stores with purely column-oriented engines that take vertical fragmentation to the extreme, and redesign the Relational DBMS architecture enabling the combination of light-weight encoding and vector processing (see Chapter 7 for details).

2.2.4 Design method

Volume and Velocity are important, but we should not forget the Variety, since in some cases, the target schema may not exist a-priori. Even when this is known, some data have a complex structure which is also highly variable and potentially evolves. Nevertheless, requiring more flexibility in the schema does not mean that we cannot still benefit from traditional design approaches. The lack of know-how to address data design results in most solutions being designed only considering performance. However, starting from the conceptual model and adopting the classical 3-phase design used for Relational databases (i.e., conceptual, logical and physical), we can consider the new challenges and features brought by NOSQL, encompassing Relational and co-Relational design altogether. We should drive the design from the conceptual schema and find a physical design coping with Variety and resilient to Variability while performance penalisation is minimised. The conceptual model should be firstly used to devise the logical schema, which can be later tuned by considering the analytical query workload to obtain the physical schema (see [HAR16]).

The **large heterogeinity** (i.e., Variety) involving some entities becomes the first limitation (e.g., dozens of products, several hundreds of services and some thousands of supplements for those services). Relational tables provide a rigid homogeneous representation of entities and we would need to either create a table for each possible *specialised* entity or a single *general* table containing the union of all attributes. This would poorly perform since deploying thousands of tables, that would be joined to produce the general entity, or creating a single table with thousands of attributes results unpractical and generates expensive queries. Furthermore, new entities constantly appear (e.g., products constantly developed and released), which would result in dynamically creating new specializations. Alternatively, this can be solved by implementing tables containing generic columns storing different attributes depending on the specialised entity. For example, the `product` table contained hundreds of generic columns of type `varchar(50)`. A row representing `productA` stores in column `C` the `product model`, while those of type `productB` use the same column `C` to store `location`). A dictionary at the application level keeps track of the mapping of each `product` column (e.g., `C`) to its real meaning (e.g., `productA` $\rightarrow$ `model`). The application is then able to understand the meaning of each column given the corresponding entity by means of the mappings. Schemaless or semi-structured databases clearly improve this solution.

[8] `https://www.mongodb.com`
[9] `https://hbase.apache.org`

Schema evolution (i.e., Variability) becomes extremely important in the context of analytics as executives constantly look for new patterns or markets and therefore ask for new data to be included in the decisional datasets. Changes might be simply ignored, resulting in data scientists spending most of their time collecting data from different sources, and cleaning and merging them by themselves prior to conduct the analysis. Reflecting such changes in the Relational model is possible, but turns out to be costly as it either requires to alter the current table (massively updating the new columns for existing instances) or create a sibling one with the same key for the new attributes. Again, considering flexible schemas can be a solution.

Indeed, during the logical design phase, we need to decide whether a given entity must be designed either following the Relational or the co-Relational model, and then it is the third physical phase that refines such decision by considering technical aspects and specificies in the implementation of concrete DBMSs. Importantly, note that deciding Relational or co-Relational to design an entity is not bound to the choice of a specific kind of DBMS, but to unveil its nature. For example, we could deploy the system either in a commercial Relational DBMS (i.e., Oracle) or in an open source co-Relational store (i.e., HBase[10] plus Hive[11]). The output of our design then hints the best storage model, but the subsequent technological instantiation and the corresponding product-oriented tuning requires a dedicated tool selection process. Thus, we can see that Relational data can be stored in NOSQL systems, and oppositely co-Relational data of semi-structured nature can be stored in a RDBMS.

For example, despite having a Relational architecture underneath, Oracle[12] supports data structures traditionally not considered Relational-like. Of special relevance for are the data types *XMLType* and *NESTED TABLE*. The former corresponds to XML data and the latter to tables embedded in other table columns. These data structures map indeed to co-Relational. Documents in Oracle can therefore be stored through the data type *XMLType*, and vertical fragments be implemented with *NESTED TABLE*. Thus, families can map to nested tables formed by two columns. The first one could be of type *VARCHAR* to refer to the column (within the family) name. The second column could be the column value and it could be of any type; we could define it as *VARCHAR* if any possible value can be stored there and then let the application understand what is inside, or more complex data structures such as *XMLType* could also be defined in order to simulate a Wide-column store with support for documents. Lastly, the *VARRAY* type allows having lists of undetermined length and therefore corresponding nested lists in the co-Relational model. Thus, Relational static entities would be designed as regular Relational tables whereas co-Relational highly variable entities can be designed through *XMLType* structures (if no vertical fragmentation is applied), and *NESTED TABLES* for entities vertically fragmented.

Let us consider now HBase plus Hive as representatives of the open-source world. HBase is a Wide-column system. Consequently, vertically fragmented entities can be naturally stored, regardless being Relational or co-Relational. Similarly, both Relational and co-Relational entities where vertical fragmentation does not apply can still be designed as single-family HBase tables. Nevertheless, benefits from using Relational structures (HBase has no global schemas and therefore embeds the schema into each instance) and document stores (column values are stored as string and parsing relies on the application level) are then lost in HBase. This cannot be solved from the point of view of the storage, but Hive can be added on top to provide a Relational view so that queries (using HiveQL) can be run as if the underlying storage was Relational. This is clearly not a true Relational implementation, but would somehow resemble it from the application point of view. HiveQL even provides also extensions like *LATERAL VIEW EXPLODE* to easily manage arrays, which would resemble sibling Relational mechanisms.

2.3 Alternative storage structures

Highly scalable DBMS designed to deal with Big Data in the Cloud neither follow the Relational model nor support declarative SQL standard, consequently being unfortunatelly tagged as NOSQL, which can be a misleading term, since it suggests there is some problem in SQL. However, that is far from reality. Having a declarative language is as good as the optimizer behind it (which is actually pretty good in RDBMS today). An important consequence of not having such declarative language is the lack of an optimizer that releases users from being concerned with too low-level efficiency issues. Instead, we should rather understand NOSQL as simply opening the door to alternative ways of storing the data, depending on their structure and use. As we have already discussed in previous sections, it

[10] `https://hbase.apache.org`
[11] `https://hive.apache.org`
[12] `https://www.oracle.com/database`

is not even true that semi-structured data cannot be stored in a supposedly rigid RDBMS. Therefore, structuredness of data should not determine their storage system.

2.3.1 NewSQL Systems

If we keep the discussion at a relatively high level, we can see that OLTP and OLAP generate contradictory (to some extent incompatible) workloads. Thus, oversimplify the storage choice, we could say that RDBMS are more appropriate for OLTP, and offer ACID transacions and support for a standard declarative query language (i.e., SQL), but have difficulties to scale out and reach high availability; while NOSQL systems on the contrary are more appropriate for OLAP-like analytical workloads, and offer easy scale out that also guarantees full availability, but relax the transactions model and fail to offer a declarative and standard query language (by now). Some systems usually known as NewSQL try to combine the best of both (e.g., CloudSpanner[13], CockroachDB[14], LeanXcale[15], SAP HANA[16], VoltDB[17]) under some conditions or constraints (e.g., using specific hardware). These systems are able to deal with Hybrid Transaction/Analytical Processing (HTAP) workloads.

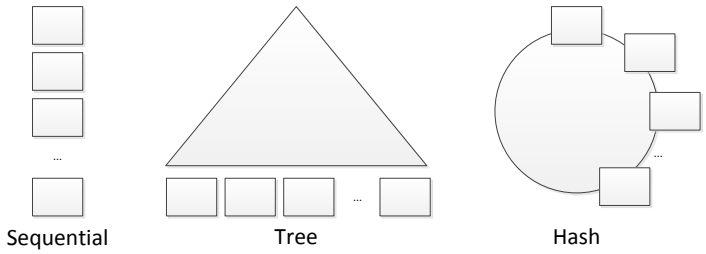


Figure 2.4: Alternative structures

2.3.2 Heterogenous Storage Systems

If we go deeper and open the options to different kinds of NOSQL, on choosing the right system for our data, we should pay attention to performance, which has nothing to do with the API offered, but rather depends on the data structure underneath. Figure 2.4 sketches three of such structures. On the first hand, if we purely require sequential scans on append-only files, the best option is a plain file, like those offered by the operating system. An example of this could be HDFS.[18] On the other hand, if we require random access but still having a good ingestion rate, we could use a Log-Structured Merge-Tree (LSM-tree, see [OCGO96]), because of being more scalable than the traditional B-tree in Relational systems. While still keeping the data sorted and potentially distributed in the cluster of machines, this structure allows quickly data ingestion. An exemplifying implementation is Apache HBase.[19] Alternatively, we find in many systems a Consistent Hash indexing structure (see [KSB+99]). In this case, data is distributed in the cluster by means of a hash function that maps every key and also machine to a ring. An exemplifying implementation is Apache Cassandra.[20]

Actually, all these data structures and techniques are not really new. NOSQL are simply modularly putting in use diverse pre-existing techniques and theories that were not available in monolithic RDBMSs.

2.3.2.1 RUM conjecture

One may think at this point that the solution to avoid any choice is to come up with a tool able to efficiently serve all possible workloads. However, such a magic wand does not exist. There are three contradictory factors in any

[13] https://cloud.google.com/spanner
[14] https://www.cockroachlabs.com
[15] https://www.leanxcale.com
[16] https://www.sap.com/products/hana.html
[17] https://www.voltdb.com
[18] http://hadoop.apache.org
[19] http://hbase.apache.org
[20] http://cassandra.apache.org

workload requirement, namely Read performance, Update performance and Memory usage (RUM). As explained in [AKM+16], "*Designing access methods that set an upper bound for two of the RUM overheads, leads to a hard lower bound for the third overhead which cannot be further reduced.*" Thus, some parts of (or at some point) our information system require to be optimized for read, others for update and others for memory use. The problem is that there cannot be any data structure that optimizes the three at once. Therefore, we need to identify our priorities and find the structure that optimizes those without hurting much the other.

An option might be to identify as well which is the most critical piece of data for us and use the same choice of system for all the data. Nevertheless, it should be obvious that this is not a good option, either. We should instead identify the workload requirements for every data entity and hence the most appropriate structure (consequently system) for it. Most probably, we will end up with contradictory requirements resulting in different systems (some HBase, some MongoDB, etc.) that need then to be interconnected to offer a single view to the final user.

2.3.2.2 Polystore sytems

Given the Variety we have to deal with, Big Data demands a shift of paradigm in data management from generic RDBMSs to more diverse and specific ad-hoc Polystore (see [TCGM17]) or Polyglot (see [SF12]) system interconnecting diverse and complex NOSQL tools (hard to design and manage). There are different Big Data processing frameworks like Apache Drill[21] and Apache Spark[22] which have drivers for different stores. However, the current situation resembles that in the 60's and 70's, when the burden of optimizing the storage and access to the data laid on software developers' skills. Today, it is even worse, because it is not specialized software engineers that use Big Data, but rather Data Scientists whose expertise must be much broader and generic. Special attention must be paid in this context to self-tuning and physical schema evolution (maybe driven by the conceptual design). Also data migration and integration management needs to be revisited to reduce the IT burden of Data Scientists, so that they can focus on analytical duties. This can only be achieved by annotating content and features with mechanisms similar to those already in use in the Semantic Web.

[21] `http://drill.apache.org`
[22] `http://spark.apache.org`

Chapter 3

Distributed Data

Cloud computing offers new possibilities that make Big Data a reality. It has definitely changed the landscape of tools we use to process the humungous amount of available data today (see [CZ14]). Thus, besides the simplicity and flexibility of not enforcing a schema, the main goal of NOSQL is to be able to manage limitless data using the power of the Cloud. Consequently, the term encompasses different kinds of distributed data management tools conceived to run in the Cloud and benefit from parallelism and commodity hardware to be able to scale (see [Cat10]), which is a great limitation in Relational Database Management Systems (RDBMS). Besides that, availability is another side effect of distribution in the Cloud, which is usually hardly achieved in RDBMS. It is true that this can be solved too with recovery mechanisms, but thousands of machines in the Cloud also facilitate it a lot.

3.1 Distributed Data Management

As already mentioned, the way to deal with Big Data is parallelizing their processing. However, before that, we need to distribute the data themselves. Managing the data in a distributed way has some specificities compared to a centralized approach that we need to study.

3.1.1 Distributed Systems

As defined in [CDKB11], a distributed system, in general (not necessarily a database), is "*One in which components located at networked computers communicate and coordinate their actions only by passing messages.* Thus, it is important to highlight that different components of the system can work concurrently (a.k.a. in parallel). This is obviously good, since allows to improve performance, but also complicates its management, mainly but not exclusively due to concurrency access, which is hindered by the lack of a global clock. Also, the more complex and more components a system has, it is easier that something fails. Ideally, when this happens, the rest of the system should be able to keep on working even if some parts cannot (maybe offering only partial functionalities).

First desired property of a distributed system is openness, as opposed to proprietary protocols and interfaces. Specific hardware/software should be avoided and platform-independent software is preferred (we can give this for granted). In general and independently of being open or not, the different components of the system can also be heterogeneous or not, but unless said otherwise, we will assume they are homogenous (i.e., all of the same kind) and lack of any autonomy in its parts to decide their design or behaviour separately from the whole. Finally, confidentiality is usually a concern in databases and is clearly compromised with distribution. Nevertheless, we will assume it is not a problem for us, either. Thus, once discarded openness, heterogeneity of components and confidentiality, the following challenges still remain in a distributed system:

Scalability: It must be able to continuously evolve to support a growing amount of tasks. This can be achieved by upgrading or improving the components (i.e., scale up). However, we will be more interested in doing it by adding new components (i.e., scale out). The latter mitigates the generation of bottlenecks, but we need then to pay special attention to the extra communication it generates between the growing number of components (specially, if we have a single coordinator). This can be solved to some extent by using

direct communication between peers (i.e., bypassing the coordinator). On the other hand, it should not be necessary to say that adding components to a system is absolutely useless if the whole workload keeps on going to a single component. Thus, load-balancing is crucial, and, ideally, should happen automatically, without human intervention.

Performance / Efficiency: As part the quality of service, it must guarantee an optimal performance and efficient processing. This is usually measured in terms of latency (i.e., response time) and throughput (i.e., requests solved per time unit). Even if they look like two sides of the same coin, they are actually contradictory. Parallelizing reduces response time, but uses more resources to do it (e.g., communication). Hence, it negatively impacts throughput unless we increase the resources to compensate. This can be mitigated, for example, optimizing network usage or using distributed indexes.

Reliability / availability: Also as part of the quality of service, it must keep performing tasks even if some of its components (either software or hardware) fail. Obviously, this is not always possible, and some functionalities might be affected, but at least a partial service could be provided. Heartbeat mechanisms that monitor the status of the components and automatic recovery mechanisms help to reduce the down time of components. Also part of the challenge is keeping the consistency of data shared by different components, since this requires synchronization, which can simply be impossible in case of network failure. This affects reliability, because the system would provide inconsistent results. Asynchronous synchronization mechanisms and flexible routing of network messages can mitigate this.

Concurrency: We could say that sharing resources is the main purpose of a distributed system. Thus, the system should provide the required control mechanisms to avoid interferences and deadlocks in the presence of concurrent requests. Once interferences happen, consensus protocols can help to solve the conflict and continue working without further consequences.

Transparency: Above all those challenges, the main one is to hide them. Users of the system should not be aware of all its many complexities. Ideally, they should work as if the system were not distributed.

3.1.1.1 Distributed Database Systems

The Encyclopedia of Database Systems, in [Tan18], defines a Distributed DataBase (DDB) as "*an integrated collection of databases that is physically distributed across sites in a computer network.*" Immediately after that, a Distributed DataBase Management System (DDBMS) is defined as "*the software system that manages a distributed database such that the distribution aspects are transparent to the users.*" We should pay special attention to three concepts in this definition: "integrated collection", "distributed accross sites in a computer network" and above the other two "distribution aspects are transparent to the users":

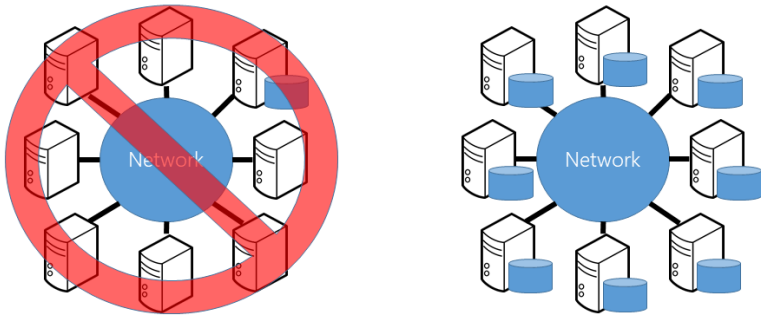


Figure 3.1: Illustration of a centralised database on a network vs real DDB

- A DDB being integrated stresses the fact that, like any database, it is something more than a disparate collection of files. Indeed, files should be somehow structured and a common access interface should be provided so that the physical location of data does matter.
- Another common mistake is to think that, despite providing access through a computer network, the database resides in only one node. If so, this approach would not very different from centralized databases and would

not pose new challenges. Oppositely, DDBMSs entail something else, and they provide an environment where data is distributed among a number of sites. Thus, LHS of Figure 3.1 is only a centralized database accessible through a network, while RHS really distributes the data. Tightly related to the previous point, note that 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 only required characteristic is that the communication between nodes is done through a computer network instead of simply sharing memory or disk.

- Finally, making distribution transparent to the user is, indeed, a massive statement entailing many consequences. Transparency refers to separation of the higher-level view of the system from lower-level implementation issues. Thus, the system must provide the mechanisms to hide the implementation details. For example, the user must be able to execute distributed queries without knowing where data are physically stored and hence, data distribution and replication must be an internal issue for the DDBMS. Similarly, the DDBMS must be responsible for guaranteeing safety properties at any moment like on dealing with update transactions, which are obviously affected by distribution, or ensuring data consistency when replication happens (i.e., synchronization between copies), or coping with node failures to maintain the availability of the system.

Distribution Transparency must guarantee data independence as well as network, fragmentation and replication transparency on top of that. Let us introduce all these one by one:

Data independence: This is fundamental to any form of transparency, and centralized DBMSs also provide it (see Section 2.2.1). Data definition occurs at two different levels: logical (schema definition) and physical. Logical data independence refers to the immunity of users applications (through views) to changes in the logical structure (i.e., the schema) of the database, whereas physical data independence, on the other hand, hides the storage details to the user.

Network transparency: In a centralized database, the only resource to be shielded is data. In a DDBMS, however, there is a second resource to be handled in the same manner: the network. Preferably, the user should be protected from the operation details of the network, even hiding its existence whenever possible. Two kinds of transparency are identified at this level: **location** and **naming** transparency. The former underlines the fact that any task performed is independent of both the location and system where the operation must be performed. The latter refers to the fact that each object has a unique name in the database irrespectively of its storage site. In absence of this, the user is required to embed the location name with the object name. Location transparency is required to have naming transparency.

Replication transparency: As briefly mentioned above, it may be interesting to replicate data over different nodes. Main reason to do so is performance, as we may avoid network overhead by accessing data locally (i.e., replication increases locality of references) and gain efficiency by using replicas to perform simultaneously some actions over different copies. Furthermore, it also provides robustness, as if one node fails, we still have the other copies to access. However, the more we replicate, the more difficult is to deal with **update transactions** and keep all replicas consistent. The update transparency refers to whether synchronizing replicas is left on the user's shoulder or automatically performed by the system. Ideally, all these issues should be transparent to the users, and they should act like if a single copy of data were available (thus, as if one object were available instead of many).

Fragmentation transparency: On fragmenting a piece of data, each fragment would be considered a different object. Again, the main reasons for fragmentation are reliability, availability and performance, as it can be seen as a way to diminish the negative aspects of replication. When datasets are fragmented, we have the problem of handling queries over a dataset to be executed over its fragments. This typically entails a translation from the *global query* into *fragment queries*. If this transparency level is provided, this translation is performed by the DDBMS, which must know the criteria used to fragment the data.

Importantly, note that all these transparency levels are incremental, as depicted in Figure 3.2. Distribution transparency implies hence fragmentation, replication, network transparency as well as data independence. From the user point of view, distribution (i.e., full) transparency (also known as query transparency) is obviously appealing, but the fact is that, in practice, it is well-known to be hard to achieve. In short, full transparency makes the

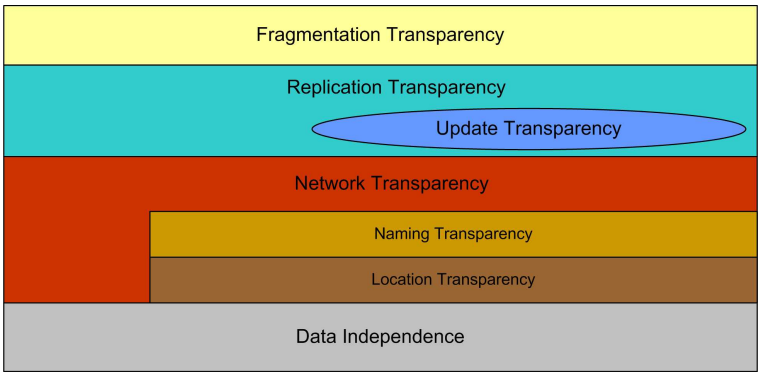


Figure 3.2: Stack of different levels of transparency

management of distributed data very difficult (in the sense that many bottlenecks and inefficiencies are introduced). For this reason, nowadays, 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.

	Autonomy	Single Schema	Query Transparency	Update Transparency
Homogeneous DDBMS	No	Yes	Yes	Yes
Tightly Coupled Federated DBMS	Low	Yes	Yes	Limited
Loosely Coupled Federated DBMS	Medium	No	Yes	Limited
Multi-database Systems	High	No	No	No

Table 3.1: Comparison of heterogeneous and autonomous DDBMS

A fair degree of transparency can be obtained in homogeneous systems. However, it is clearly compromised if we consider heterogeneous and autonomous systems. As summarized in Table 3.1, paying attention to how autonomous they are, they range from tightly coupled federated systems (i.e., having some autonomy and a single schema), to multidatabase systems (i.e., a set of completely autonomous and heterogeneous databases not even offering a common schema), through some kind of loosely coupled federated systems (i.e., also with some autonomy but not necessarily offering a single schema). In all these cases, true update transparency is impossible and only homogeneous monolithic DDBMS and tightly coupled federated DBMS can offer query transparency through schema information.

An **Extended ANSI/SPARC architecture** is necessary to manage distributed schemas. These schemas are related by means of mappings, which specify how data specified at a level can be obtained from those specified at another level. They are already implicit in the centralised ANSI-SPARC architecture in Figure 2.2, and are the ones providing the means for data independence; those between the external and conceptual schemas (a.k.a. view definitions) provide logical data independence, whereas those between the conceptual and internal schemas provide physical data independence. Such mappings are hidden in the database catalog available in any DBMS.

However, the original ANSI/SPARC architecture does not consider distribution and it must be consequently adapted in order to provide distribution transparency. As shown in Figure 3.3, in presence of distribution, a global conceptual schema is needed to define a single logical database (i.e., the conceptual view of the organization). This conceptual schema corresponds to the very same idea behind the original ANSI/SPARC architecture, but the crucial difference is that the database is composed of several nodes (instead of just one) and this general schema provides together with logical independence also network, replication and fragmentation independence (i.e., provides distribution transparency). Furthermore, every component also defines a local conceptual schema and an internal schema. Again, mappings between the external schemas and the global conceptual schema, as well as those between the global and local conceptual schemas are needed in order to translate a global query in terms of local ones. Such mappings are stored in the global catalog. Out of all these mappings, the ones between the global and local conceptual views are known as *fragmentation and allocation schemas*. It is important to notice that local catalogs are still needed to solve mappings between the local conceptual schema and the internal schema. Finally, note that in this extended schema architecture we can talk about the global unique conceptual view (which

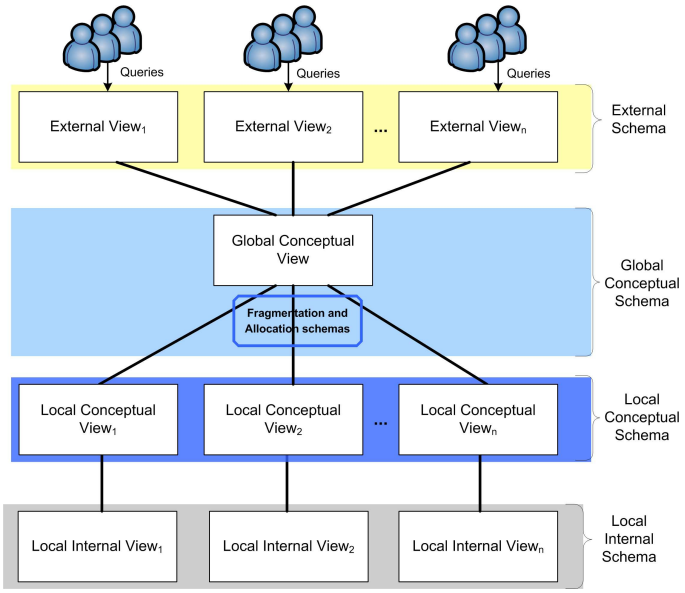


Figure 3.3: Extended ANSI/SPARC Schema Architecture for DDBMSs

hides distribution and where each data object has a unique name), and local views (which are aware of naming and distribution).

An **Extended DBMS Functional Architecture** is therefore necessary to solve the new transparency requirements (schemas are just passive means to do it). A DDBMS is composed by a set of functional components to successfully manage the database. When running in a computer, DBMS interact with applications through their interface layer at the highest abstraction level, whereas they communicate at the lowest abstraction level with the operating system through their communication layer. In between, a large number of components interact to form the DBMS as a whole. Figure 3.4 sketches the main components of a centralized DBMS.

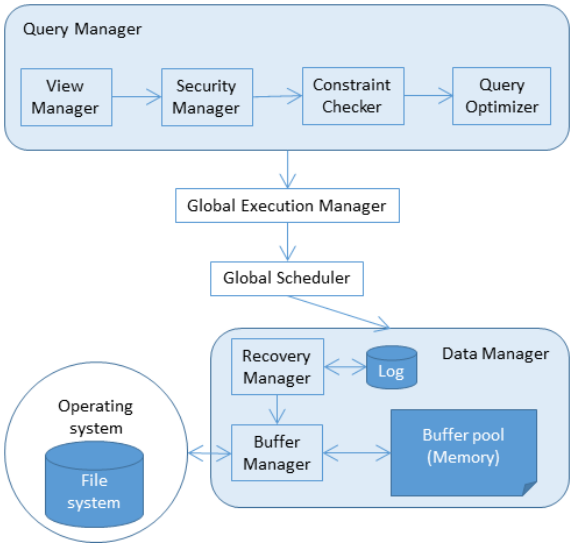


Figure 3.4: Functional Architecture of a centralized DBMS

Users pose queries over the database through the interface layer and, subsequently, these queries reach the query manager component. The query manager contains the view manager (in terms of the ANSI/SPARC architecture, a query is posed over the external schema, and the view manager is responsible for rewriting the input query in terms of the conceptual schema making use of the corresponding mappings), the security manager (responsible

for performing the corresponding authorization checks), the constraint checker (responsible for guaranteeing that integrity constraints are preserved in case of updates) and the query semantic, syntactic and physical optimizers, responsible for, respectively, performing semantic optimizations (i.e., transform the input query into an equivalent one of a lower cost), generate the syntactic tree (in terms of Relational algebra operations) and the physical access plan to data. Next, the execution manager launches the different operators in the access plan in order (for example, a selection operator that must be solved prior to solve the subsequent join). Then, for each query operator executed, it is also responsible for building up the corresponding results. The scheduler deals with the problem of keeping the database in a consistent state even when concurrent accesses occur. In short, it preserves isolation (I) property of ACID transactions. It sits on top of the recovery manager that is responsible for preserving the consistency (C), atomicity (A) and durability (D) properties, which, in turn, sits on top of the buffer manager, responsible for bringing data to main memory from secondary storage, and vice-versa. Thus, the buffer manager communicates with the operating system.

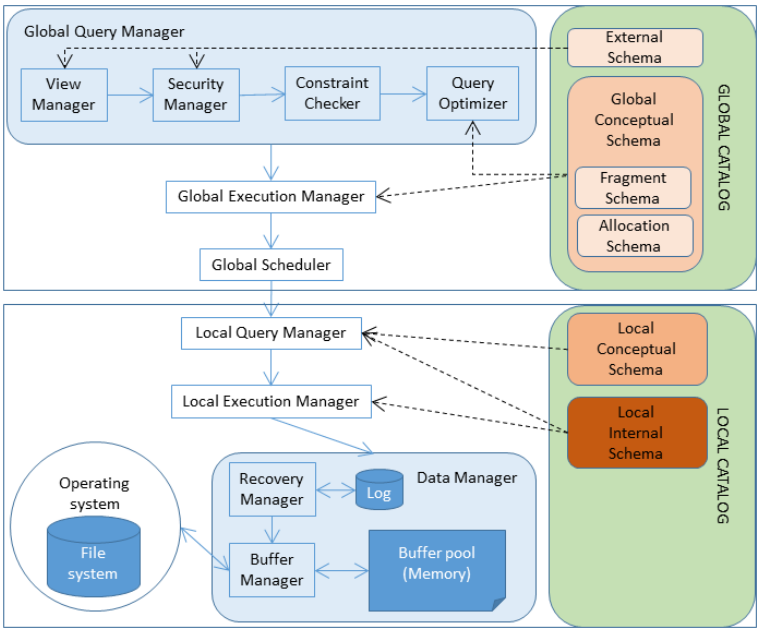


Figure 3.5: Functional Architecture of a DDBMS

To be able to deal with a DDB, there are several refinements to be done. As depicted in Figure 3.5, there are two well-differentiated stages. The first one corresponds to modules cooperating at the global level discussed by the extended ANSI/SPARC architecture, whereas the second one corresponds to those cooperating at the local level. Thus, in the first stage, the data flow is transformed and mapped to the lower layers by dealing with a single view of the database (once the external schemas are resolved), and the distribution transparency. When queries reach the second stage, fragmentation, distribution, and replication have already been solved. More specifically:

- The global query manager contains the view manager, the security manager, the constraint checker and the query semantic and syntactic optimizers. All the previous behave like in a centralized DBMS except the syntactic optimizer that is extended to consider data location (by querying the global data catalog). Finally, the physical optimizer is also extended for exploiting the metadata stored in the global catalog and decide which node executes what (according to replicas and fragments available and communication cost over the network) and using which execution strategy (e.g., to perform a join). New optimization criteria happen to be relevant for distributed query execution, such as minimizing the size of intermediate results to be sent over the network and exploiting parallelism as much as possible.

- Next, the global execution manager inserts communication primitives in the execution plan and coordinates the execution of the pieces of query in the different components. Again, it will be responsible for building up the final result from all the query pieces executed in a distributed way.

- Then, the global scheduler receives the global execution plan produced in the previous component and

distributes tasks between the available sites guaranteeing isolation between different users.

- Next, each node receives its local piece of query and, by retrieving the corresponding mappings from its catalog, generates a local access plan by means of the local query manager. The local query manager behaves similar to a centralized query manager and has, among several duties, to decide which structures are used to optimize data retrieval. Subsequently, the data flows as in a centralized version of a DBMS, and goes through the execution, recovery and buffer managers.

3.1.1.1.1 Cloud Databases At this point, it is crucial to realize that any DBMS (actually, any piece of software) can be installed in a virtual machine and hence run in the Cloud. However, there is not really any benefit in doing this, and such piece of software would have exactly the same capacity as if executed on premises in a physical machine (or even less). What really makes the difference is designing completely new software tools (a.k.a. Cloud Databases) that can benefit from running in the Cloud and make use of its many innovative features (i.e., elasticity, ubiquity and availability). These new tools depart from the traditional disk-based centralized architecture to focus on the best way to parallelize data processing.

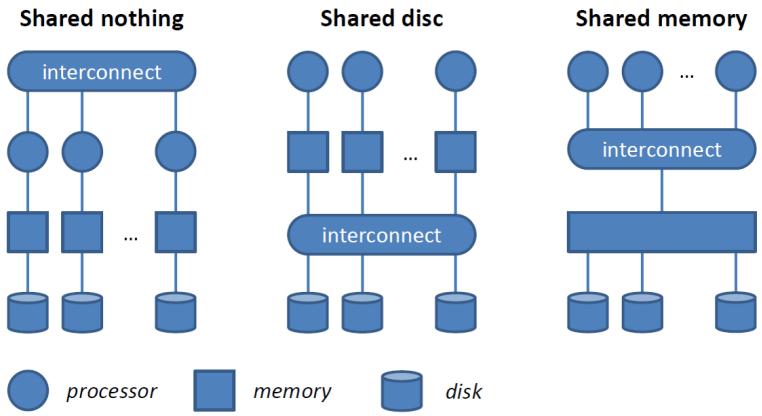


Figure 3.6: Parallel architectures, by Daniel Abadi

The first thing we should be aware of when dealing with Big Data is that systems must be able to scale out by adding new machines to our cluster or Cloud (as opposed to scale up, which upgrades or extends hardware inside the same machine). Figure 3.6 depicts the tree alternative parallel hardware architectures. From right to left, the first option is "Shared memory", which has a bus interconnecting the different processors to a common memory, which in turn can host data coming from multiple disks. Obviously, this memory bus becomes a bottleneck and requires strict synchronization between the many processors, which compromises scalability (usually restricted to a single machine). The second, and much more popular, option is "Shared disk", comprising a set of tightly coupled multiprocessor systems that still run a single copy of the operating system[1]. In this case, even if there are multiple machines, the OS is unique so synchronization has to happen at this level, blocking whatever happens above it. Finally, the most popular option is "Shared nothing", which interconnects many completely independent machines through the network. Each machine has its own disk, memory and obviously processor, which work independently, but at the same time are exposed to the others. It is easy to see that the latter is the most scalable and coincides with the architecture of the Cloud.

It is easy to imagine that moving from a fixed capacity machine on premises to a supposedly infinite capacity pay-per-use Cloud has many consequences (see [Cat10]) related to the challenges of distributed systems above:

a) First of all, related to scalability, it would be absolutely useless to have thousands of machines if the execution is limited to a single machine. Therefore, it is crucial that execution can be (automatically) spread in as many machines as possible (a.k.a. horizontally scale or scale out).

[1] https://en.wikipedia.org/wiki/VMScluster; https://en.wikipedia.org/wiki/IBM_Parallel_Sysplex

b) From a performance/efficiency point of view, since we are talking about data intensive applications, it does not make any sense to think about distributed execution if data is centralized. Thus, efficiently scaling out entails data distribution and replication, which in the end builds on fragmentation techniques.

c) Data management is a complex task, and even more in a distributed environment. It would not be realistic to expect that developers control all the knobs that traditional RDBMS have. Thus, Cloud databases take the approach of simplifying things as much as possible, and just use as efficiently as possible distributed memory and indexing mechanisms to parallelize execution. Beyond that, speed up basically relies on massive replication and the brute force of parallelism. As a side effect, replication and parallelism contribute to improve reliability and availability.

d) The stopper of parallelism are synchronization barriers. Some of those are unavoidable due to the nature and needs of algorithms, but others can be circumvented by relaxing some desirable characteristics. This is the case of concurrency control, which generates contention (i.e., some execution threads need to wait for others to release the resources before proceeding). The traditional ACID transaction model is well known for the overhead generated due to the typical locking mechanisms that it entails. Thus, from the point of view of concurrency, Cloud Databases relax such strong consistency and define the new much weaker concept of *eventual consistency*.

e) If we want to find a word defining Cloud Databases, this must be "simplicity", and it is through this simplicity that they reach the required efficiency. Thus, a simplistic call level interface or protocol is typically provided to manage data, which is easy to learn and use, but puts the optimization burden on the shoulders of the developers. As a counterpart to the efficiency obtained this way, some transparency is compromised. The schemaless nature of most of these systems complicates even more the possibility of enjoying a declarative query language like SQL (automatic translation from a declarative syntax to a concrete procedure is impossible today if the schema and data types are not known a priori).

f) Beyond previous challenges, in a Cloud with literally thousands of machines, it is simply impossible to install and fine tune all of them one by one manually. Moreover, benefitting from elasticity (enabling or disabling resources depending on the concrete needs at a given time), means that switching on and off a machine or service must be immediate. Thus, it is mandatory the setting up of hardware and software is quick and cheap.

g) The new business model where users do not pay the ownership of hardware and software, but instead pay only depending on how much they use it, means that providers have to change the way of managing things. On the one hand, there is no ownership of the software, and on the other users want to start using it immediately (like they provision a virtual machine).[2] Thus, appears the concept of multi-tenancy,[3] where the same hardware/software is shared by many tenants.[4] A prominent examples of multitenant Cloud database are Amazon SimpleDB,[5] Google Bigtable,[6] and Google Spanner.[7] Obviously, this requires some mechanisms to efficiently manage the sharing and actually benefit from it. Multi-tenancy can be used, for example, to maximize the usage of resources by properly scheduling all users to distribute the workload in geography and time.

h) The fact of having multiple tenants and users in general, together with new agile software development methodologies means that rigid pre-defined schemas are not appropriate for databases. Instead, there is the need of gaining flexibility and the ability to dynamically add new attributes, for example.

By this way of tackling the distribution challenges, Cloud databases provide new solutions to address the three V's of Big Data by (i) distributing data and processing in an extensive shared cluster (typically of commodity machines) and (ii) by introducing alternative and simple database models. Indeed, the way to overcome RDBMS limitations is for Big Data architectures to build on NOSQL tools conceived to live in the Cloud. Indeed, most NOSQL systems distribute data (i.e., fragment and replicate them) in order to parallelize their processing while

[2] `https://snarfed.org/amazon_simpledb_thoughts`

[3] `https://www.youtube.com/watch?v=t_gyloi_xTE`

[4] A tenant is simply the person or entity that pays the rent for the use of the land or a building (the word is then borrowed for Cloud users with a similar meaning).

[5] `https://aws.amazon.com/simpledb`

[6] `https://cloud.google.com/bigtable`

[7] `https://cloud.google.com/spanner`

exploiting the data locality principle, and hence avoiding data shipping from one machine to another. However, for this to happen both sides firstly providers, but also designers, administrators and developers still need to be aware of the challenges this poses to their management:

Provider From the platform provider point of view, the difficulty is of course the potentially high number of tenants, but also the unpredictability of their workloads characteristics. Firtsly, popularity of tenants can change with even flash crowds using their services, which would transitively impact the Cloud database. But it is not only the number of final users that change, but also the activities they realize (some being more or less demanding for the database underneath). Thus, the Cloud database provider has to implement some mechanisms to be able to deal with all that variety and variability in the workload. Firstly, the most basic feature is to have metadata about tenants, including which features they enable). These metadata should allow self-management of the databases, since it is simple unfeasible to manually configure all served databases. For similar reasons, but also to guarantee high availability, the system should tolerate failures, and also offer some self-healing mechanisms, if possible. Finally, still from the perspective of offering high quality of service, 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. In the line of thoughts, some tenants may require to shrink the resources used, too. Thus, data underneath should not only be more and more distributed, but have some more elastic behaviour where they can also be compacted back.

Tenant/user From the platform user point of view, we can identify four big challenges that need to be carefully considered in the presence of distribution:

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

II. **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 only 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 or developpers.

III. **Transaction management** results to be specially hard and expensive in distributed environments. Distributed recovery and concurrency control mechanisms and protocols exist, but the point is to find a trade-off between the security they guarantee (in terms of availability and consistency) and the performance impact they have. Specially relevant in this case is the management of replicas, which are really expensive to update, but reduce query latency and improve availability.

IV. **Query processing** must be as efficient as possible, this being one of the main aspects of NOSQL tools. To this end, parallelism should benefit from data distribution, without incurring in much communication overhead, which can be reduced by replicating data.

3.1.2 Challenge I: Data Design

It must be obvious that in data intensive applications, distributing the processing is not enough. We also need to distribute the data. One of the main reasons why DDBMSs are considered to be a solution to manage large-scale data repositories, is that they facilitate data locality (i.e., data is local to the process), which together with parallelism are seen as an efficient manner to put into practice a divide-and-conquer approach. However, to really benefit from that, DDBs must be designed to maximize such locality and reduce resource contention and communication overhead. As discussed in previous section, data locality can be improved by means of fragmentation. Nevertheless, this is not that simple, because it is extremely hard to know a priori where a given task will be executed (this depends basically on where the resources are available at the time of execution). Thus, there is not any way to have the data ready at a given place always ready to the right task. A way to mitigate this problem is replication (i.e., having copies of data at different places). This offers alternatives to the execution of tasks, but comes at the cost of maintenance. It is important to clearly distinguish between fragmenting and replicating those fragments. The former is related to the problem of finding the proper working unit of the distributed system, whereas the second one is related to allocation of that work.

3.1.2.1 Data Fragmentation

Data fragmentation deals with the problem of breaking datasets into smaller pieces and thus, decreasing the working unit in the distributed system.[8] There are many reasons to fragment (e.g., privacy/confidentiality), but the main idea is that the dataset as a whole is not a good processing unit.

Traditionally, data fragmentation has been useful to reflect the fact that applications and users might only be interested in accessing different subsets of the whole schema or even different subsets of the same dataset. For example, consider the case of a user accessing a database through a view defined as a selection over a dataset `R`. Certainly, this user will never access any data instance out of this subset. In general, applications and users will only access a subset of the datasets available in the database or, more specifically, a certain subset of some subsets. Without considering data fragmentation we have two options. Either placing the dataset at a single node (and thus, increasing the remote accesses and producing a potential bottleneck) or replicating it at every node where the dataset might be needed. However, different subsets are naturally needed at different nodes and it makes fully sense, from a performance point of view, to allocate fragments likely to be used by a task in the node where this task runs. This is what we know as data locality.

Thus, the same concept can also be used to deal with large databases, as it increases the degree of concurrency and facilitates parallelism, minimizing the communication overhead through the network, and avoiding unnecessary replication. Fragmenting a dataset may look as simple as finding alternatives to decompose it into smaller pieces. However, it can increase distributed joins and difficult integrity checking (even in some simple cases, according to where fragments are allocated) if two attributes with a dependency are split across fragments. Finding the right way to do it entails special difficulties in the presence of conflicting requirements that do not allow to come up with mutually exclusive fragments.

Clearly, we have two main fragmentation approaches: horizontal and vertical. In the first case, a selection predicate is used to create different fragments and, according to an attribute value, place each row in the corresponding fragment. Oppositely, in case of vertical fragmentation, different projections of the dataset are carried out, creating each one of them a different fragment (as will be justified later, we need to place the identifier at each vertical fragment). In general, different requirements may lead to different fragmentation strategies over the same dataset. Thus, a third option combining both, known as hybrid fragmentation, arises as an alternative. This is nothing else than nesting horizontal and vertical fragmentation strategies. In other words, further fragmenting fragments produced by a previous fragmentation.

Next subsections focus on horizontal and vertical fragmentation, and discuss two main aspects: the degree of fragmentation we may achieve and the correctness rules of fragmentation. The first aspect is related to up to which extent we are interested in fragmenting a dataset (in other words, how many subsets we want to produce), whereas the second one is related to guarantee the correctness of the fragments. Thus, a fragmentation is correct if we can guarantee completeness, disjointness and reconstruction:

- Completeness: Given a dataset `R` and a set of fragments, any data item (either a tuple or a set of attributes) of `R` can be found in, at least, one fragment. Thus, data is not lost when fragmenting.
- Disjointness: Given a dataset `R` and a set of fragments, there is no redundancy (i.e., unnecessary repetitions) in the defined fragments.[9]
- Reconstruction: Given a dataset `R` and a set of fragments, the original dataset can always be reconstructed from the fragments by means of Relational operators.

We do not discuss specifically hybrid fragmentation because corresponding to a nested combination of both horizontal and vertical strategies, it is correct if all the subsequent fragmentation strategies applied are independently correct.

3.1.2.1.1 Horizontal fragmentation A horizontal fragmentation partitions a dataset along its rows. The way to define each fragment is by means of predicates (i.e., selections over any attribute). Table 3.2 exemplifies a horizontal fragmentation, where each fragment contains a subset of the rows of the dataset.

Formally, a dataset R is horizontally fragmented in n fragments by means of a *fragmentation predicate*: $R_i = \sigma_{F_i}(R), 1 \leq i \leq n$, where F_i is the fragmentation predicate that defines fragment R_i. Typically, we represent each

[8] We use the term partitioning, instead, to refer to a dataset that is broken into small pieces which are not distributed over a network.

[9] Data replication must be considered a posteriori as an independent task to data fragmentation, in the allocation stage.

pno	name	city	country	budget	category	productivityRatio	income
1	p1	Barcelona	Spain	40,000	administration	0.1	3,000
2	p2	Barcelona	Spain	10,000	administration	0.5	36,000
3	p3	Barcelona	Spain	5,000	financial	0.8	1,000
4	p4	Frankfurt	Germany	100,000	tv	0.8	70,000
5	p5	Glasgow	UK	450,000	financial	0.6	240,000
6	p6	Glasgow	UK	2,000	financial	0.6	1,000
7	p7	Glasgow	UK	1,000	financial	0.3	2,000
8	p8	Portland	USA	30,000	culture	0.2	2,000
9	p9	Hong Kong	China	7,000	others	0.1	10,000
10	p10	Hong Kong	China	10,000	financial	0.9	40,000

Table 3.2: Example of Horizontal Fragmentation

fragment by the predicate used to create it. In our example: $HF_1 : city = Barcelona$, $HF_2 : city = HongKong$, $HF_3 : city = Portland$, $HF_4 : city = Frankfurt$, and $HF_5 : city = Glasgow$.

The first issue to address is deciding when we should horizontally fragment a given dataset. In general, a distributed system benefits from horizontal fragmentation when it needs to mirror geographically distributed data (each node mainly accesses data related to it), to facilitate recovery and parallelism, to reduce the depth of indexes (as each fragment will have its own index; thus, it increases the number of indexes but reduces their size) and to reduce contention.[10] In our example, each fragment is obviously smaller than the whole dataset. Thus, an index over the identifier (`pno`) would result in five different indexes (one per fragment) but smaller in size (as each index will only contain entries for the rows in that fragment). Contention is clearly reduced, as several mutually exclusive users, who work on different nodes, can access different fragments simultaneously and no conflicts will be caused. Furthermore, queries over the whole dataset can be resolved by means of parallelism.

Next, we should decide up to which extent we should fragment. Note that fragmentation can go from one extreme (no fragmentation at all) to the other (placing each row in a different fragment). Furthermore, we need to know which predicates (over which attributes) are of interest in our database. To address this issue we should check those predicates used by the users (or applications) accessing the database. As general rule, we can use the claim that the 20% most active users produce 80% of the total accesses to the database. Thus, we should focus on these users to determine which predicates consider in our analysis.

Once we have identified the fragmentation predicates, we must guarantee their correctness:

- Completeness: The fragmentation predicates must guarantee every row is assigned to, at least, one fragment. As long as the fragmentation predicates are complete, the final fragmentation is guaranteed to be complete as well.
- Disjointness: The fragmentation predicates must be mutually exclusive. In other words, one row must be placed in at most one fragment. This is usually referred as the minimality property for horizontal fragments.
- Reconstruction: The union of all the fragments must constitute the original dataset. Thus, $R = \bigcup R_i$, for $1 \leq i \leq n$.

For example, the fragmentation strategy proposed in table 3.2 satisfies these properties: the fragmentation predicates are complete (as soon as we have considered all values for the `city` attribute), disjoint (being an equality, we know that an attribute cannot take two different values) and the original dataset can be reconstructed by means of the union operator.

Derived horizontal fragmentation The previously discussed horizontal fragmentation (also known as primary horizontal fragmentation) only considers one dataset at a time. However, any dataset is related to other datasets and seemingly, when querying a database schema we may use such relationships intensively. For example, suppose a dataset R related to a dataset S through a many-to-one relationship not accepting NULL values in the R-end (implemented as a foreign key - primary key constraint). Furthermore, suppose now that these datasets are usually

[10] Contention occurs when we are denied to access a database resource, because of conflicts with other users, normally as a consequence of concurrency control techniques.

queried together by joining them through the foreign key - primary key relationship. Since horizontal fragmentation tries to maximize data locality, it seems rather clear that both datasets are candidates to be placed in the same node.

To apply this strategy we will identify a *member* and an *owner* dataset (i.e., R and S, respectively). These role names are simply a notation issue (they try to highlight the fact that a member is characterised by an object and hence, somehow dependent on it), but they are relevant since the owner decides how to fragment the member. For example, it would be interesting to fragment the `employee` dataset according to `project`. In this case, `employee` would be the member and `project` will be the owner, and we would join them through the `projectNo` (foreign key) - `pno` (primary key) relationship.

As a general rule, derived horizontal fragmentation is of interest when the member fragments need to be combined with the owner fragments with matching join keys. In other words, when the member dataset is clearly dependent on the owner one, according to the database queries. If this is the case, we proceed as follows. Let us suppose the owner dataset is already fragmented in n fragments (S_i) and we want to fragment the member dataset R regarding the owner dataset, by means of a relationship `r`. The derived horizontal fragmentation is defined as: $R_i = R \ltimes S_i$, $1 \leq i \leq n$, where $\ltimes$ stands for a left-semijoin.[11] We are considering the joining attributes to be the attributes of R and S in `r`.

Note that it may happen that the member and owner datasets are related by means of more than one relationship. If this is the case, we should apply the following criteria to decide among the available relationships:

a) The fragmentation more used by users / applications (i.e., which subset is closer to what users and applications use), attending to the frequency of queries.

b) The fragmentation that maximizes the parallel execution of the queries.

Finally, in order to consider a derived horizontal fragmentation to be complete and disjoint, two additional constraints must hold on top of those discussed for the horizontal fragmentation:

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

Both, the primary horizontal fragmentation and the derived horizontal fragmentation strategies aim at maximizing data locality. However, the first considers each dataset per se, whereas the latter also considers the relationships between datasets.

pno	name	city	country
1	p1	Barcelona	Spain
2	p2	Barcelona	Spain
3	p3	Barcelona	Spain
4	p4	Frankfurt	Germany
5	p5	Glasgow	UK
6	p6	Glasgow	UK
7	p7	Glasgow	UK
8	p8	Portland	USA
9	p9	Hong Kong	China
10	p10	Hong Kong	China

pno	budget	category	productivityRatio	income
1	40,000	administration	0.1	3,000
2	10,000	administration	0.5	36,000
3	5,000	financial	0.8	1,000
4	100,000	tv	0.8	70,000
5	450,000	financial	0.6	240,000
6	2,000	financial	0.6	1,000
7	1,000	financial	0.3	2,000
8	30,000	culture	0.2	2,000
9	7,000	others	0.1	10,000
10	10,000	financial	0.9	40,000

Table 3.3: Example of Vertical Fragmentation

3.1.2.1.2 Vertical fragmentation Vertical fragmentation partitions the dataset in smaller subsets by projecting some attributes in each fragment. Consider Table 3.3 that exemplifies a vertical fragmentation. Each fragment contains a subset of the dataset attributes but notice that both of them contain the identifier (this will be justified later).

Formally, a dataset R is vertically fragmented in two pieces by means of projections: $R_1 = \pi_{A_1,\ldots,A_j}(R)$, $R_2 = \pi_{A_1,A_j,\ldots,A_m(R)}$, $1 < j \leq m$, where m is the number of attributes in `R`, and A_1 is the identifier of R.

[11] A left-semijoin pairs those tuples in R for which there is at least one tuple in $\mathtt{S}_i$ with matching joining key.

Similar to the horizontal fragmentation strategy, we first address the issue of determining if a vertical fragmentation suits our needs. Vertical fragmentation has been traditionally overlooked in practice, since, many times, it worsened insertions and update times of transactional systems (for years, the most critical problem). However, with the arrival of read-only workloads, like in decisional systems (where the user is only allowed to query data), this kind of fragmentation arose as a powerful alternative to decrease the number of attributes to be read from a dataset. This is clear to see with an extreme scenario. Consider a dataset Z with $1,000$ attributes, where each attribute requires 10 bytes. During our analytical tasks, we are interested in querying $100,000$ tuples in average, but only 10 attributes out of the $1,000$ are involved in the query. For each query we approximately read $100,000 \cdot 1,000 \cdot 10$ bytes ($\approx 10^9$ bytes), when, indeed, we only need to read $100,000 \cdot 10 \cdot 10$ ($\approx 10^7$ bytes).

In general, vertical fragmentation improves the ratio of useful data read (i.e., we only read relevant attributes) and, similar to horizontal fragmentation, it also reduces contention and facilitates recovery and parallelism. As disadvantages, note that it increases the number of indexes (all of the same size), worsens updating / insertion time and, in principle, increases the space used by data (as the primary key is replicated at each fragment). In our example, we have two different fragments, which can be represented by the set of attributes they project. VP_1: {`pno, name, city, country`} and VP_2: {`pno, budget, category, productivityRatio, income`}. This fragmentation is complete as both fragments contain all the attributes in the dataset. Despite the repetition of the identifier, this cannot be avoided (to allow reconstruction) and is thus considered non-redundant. Finally, to guarantee reconstruction, the identifier (`pno`) has been replicated at each fragment to be used in the corresponding join.

In general, deciding how to group attributes in each fragment is not obvious at all. The information required to decide it is:

a) Data characteristics

- Set of attributes
- Value distribution for all attributes

b) Workload (set of important/critial queries)

- Frequency of each query
- Access plan and estimated cost of each query
- Selectivity of each predicate

Nowadays we can benefit from well-known approaches like attribute splitting clustering to group attributes likely to be read together. In order to find the best choice, the idea is to start considering attributes isolated and then propose potential groups to be stored together. We should follow three phases:

I) Determine primary partitions (i.e., subsets of attributes always accessed together).

II) Generate a disjoint and covering combination of primary partitions, which would potentially be stored together.

III) Evaluate the cost of all combinations generated in the previous phase (based on the data characteristics and workload information).

Summing up, horizontal fragmentation mirrors geographically distributed data and boosts data locality both for querying and inserts / updates. Oppositely, vertical fragmentation better supports read-only workloads, like in decisional systems, and drastically improves the ratio of useful data read (only relevant attributes are read out of the whole set of attributes of the dataset). Vertical fragmentation is brought to the extreme in column-oriented database management systems, which store data column-wise rather than by row-wise. These systems have been shown to be extremely useful to support decisional systems and nowadays we can find successful commercial proprietary implementations such as Vertica[12] or open source like MonetDB[13] (see Chapter 7).

[12] `https://www.vertica.com`
[13] `https://www.monetdb.org`

3.1.2.2 Data Allocation

Once the database is fragmented, we must decide where to place each fragment. Given a set of fragments and a set of sites on which a number of applications/tasks are running, we aim at allocating each fragment such that some optimization criteria are met (potentially subject to certain constraints, too). Usually, the optimization criteria are defined according to two different features:

- Minimal cost: Which is a 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, the cost of updating each fragment F_i at all sites it is replicated, and the cost of data communication. The allocation problem places each fragment at one or several sites in order to somehow minimize this (e.g., we could take their weighted sum).

- Maximal performance: In this case, we aim either at minimizing the response time (given a set of queries) or maximizing the overall throughput.

This problem is known to be NP-hard and the optimal solution depends on many factors. For example, the location in which each query originates, the query processing strategies (e.g., join methods), the network latency, etc. Furthermore, in a dynamic environment the workload and access patterns may change and all these statistics should be always available in order to find the optimal solution. Therefore, the problem is typically simplified with certain assumptions (e.g., only communication cost is considered) and, typically, some simplified cost models are built so that any optimization algorithm, which in general produce sub-optimal solutions, can be adopted to solve it. Game-theory and economics techniques have proved to be useful when looking for optimization algorithms to tackle the data allocation problem.

3.1.2.3 Replication

Relevantly, note that the same fragment might be placed in several nodes and thus, replication is an issue to be addressed at this point as a generalization of allocation in more than one place, and not earlier. In general, replication must be used for reliability and efficiency of read-only queries. On the one hand, several copies guarantee that in case a copy fails, we can still use the others. On the other hand, replication can also be seen to improve data locality since it offers more possibilities to place tasks. Nevertheless, updating /inserting data in a replicated copy takes more time, and synchronizing such writings may not be trivial. As result, consistency of the copies may be affected.

If there are many copies of the data, but we only update one of them, depending on where the corresponding task is executed its result will be different (as the data are). Thus, replicating fragments improves the system throughput but raises some the issue of consistency and update performance. Yet, in highly distributed systems (thousands of nodes spread around the World), replica management becomes crucial. On the one hand, we want to keep all the copies of data consistent, but this requires many updates (one per copy) and synchronization efforts. On the other, we want to spend as little time as possible updating data, but this does not mean we do not want to have replicas. For this reason, the degree of replication must be a trade-off between performance and consistency.

3.1.3 Challenge II: Catalog management

The same design problems and criteria can be applied to the catalog. The only difference being that instead of containing data, it stores metadata. This requires two important considerations. On the one hand, metadata is much smaller that data and consequently easier to manage. On the other, optimizing performance is much more critical, since accessing this metadata is a requirement for any operation in the system.

In general, many decision are already made by the architects of the system, and only few options can be parametrized on instantiating it. For example, typically metadata in DDBMS are fragmented depending on whether they are global or local. Global metadata are allocated in the coordinator node, while local metadata are distributed in the different workers. However, a typical choice we can make in many NOSQL systems is having a secondary copy of the coordinator (a.k.a. mirroring) that takes the control in case of failure. We only need to be aware that enabling such redundancy will obviously consume some resources.

3.1.4 Challenge III: Transaction management

The cost of transferring data and the chances of communications failure are really high in the Cloud, specially when we are considering Wide Area Networks (WANs). This affects transactions in the sense that compromises all ACID properties if we keep on allowing modifications during network cut. Atomicity is not guaranteed, since only changes in client's side of the cut are applied. Because of the same reason, consistency is violated, since both sides of the cut cannot be synchronized. Finally, durability is also compromised, since some changes committed during the cut may be simply overwritten with a late synchronization after solving the network cut. Thus, in year 2000, the following conjecture (two years later proved to be true) was proposed:

> *We can only achieve two of Consistency, system Availability, and tolerance to network Partition.* – CAP Theorem, Eric Brewer

Consistency is used in this theorem in the sense that all users see the same data at the same time (the access point to the DDBMS does not matter). From some perspective, the word is missused and should be replaced by the more appropriate "convergence", in the sense that different replicas in different sites converge to the same value.[14] Availability means that if some nodes fail or are not reachable, the others can keep on working. Finally, tolerance to network partition means that even if some messages are lost, the system can continue operating. In larger distributed-scale system (i.e., thousands of computers working collaboratively), network partitions are given for granted. Thus, we must choose between consistency and availability: Either we have an always-consistent system that becomes temporally unavailable, or an always-available system that temporally shows some inconsistencies.

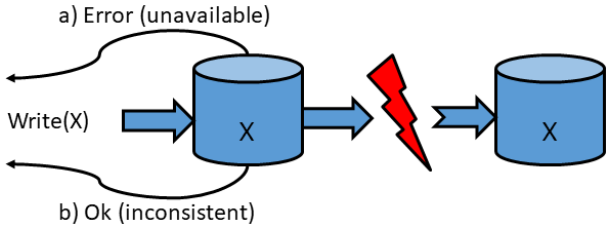


Figure 3.7: Schematization of the CAP theorem

If we give for granted connectivity loss or delay (which is quite common in Cloud environments, and is only guaranteed not to happen in non-distributed environments), two options are left (schematized in Figure 3.7):

A) Strong consistency: Replicas are synchonously modified and guarantee consistent query answering. However, the whole system will be declared not to be available in case of network partition.

B) Eventual consistency: Changes are asynchronously propagated to replicas so answer to the same query depends on the replica being used. Then, in case of network partition, changes will be simply delayed.

Basically, there are two choices that generate four alternative configurations for replica synchronization management:

Primary-Secondary versioning: We can distinguish a "primary" (a.k.a. master) copy or not. If so, this is the only one that can be directly updated by users (all the others simply mirror it). Alternatively, the users can directly update any available copy. Anyway, it is the system that guarantees that all copies have the same data (users are only required to update one copy).

Eager-Lazy replication: Users can receive confirmation of their updates after the first copy has been updated, or after all copies have been updated. In the latter case, we talk about "eager" replication (as the system needs to do it ASAP to confirm it to the user); while in the former, we talk about "lazy" replication (since the system is not urged to propagate changes, because they have already been confirmed to the user anyway).

These two choices give rise to the four alternative configurations sketched in Figure 3.8.

[14] `https://pathelland.substack.com/p/dont-get-stuck-in-the-con-game`

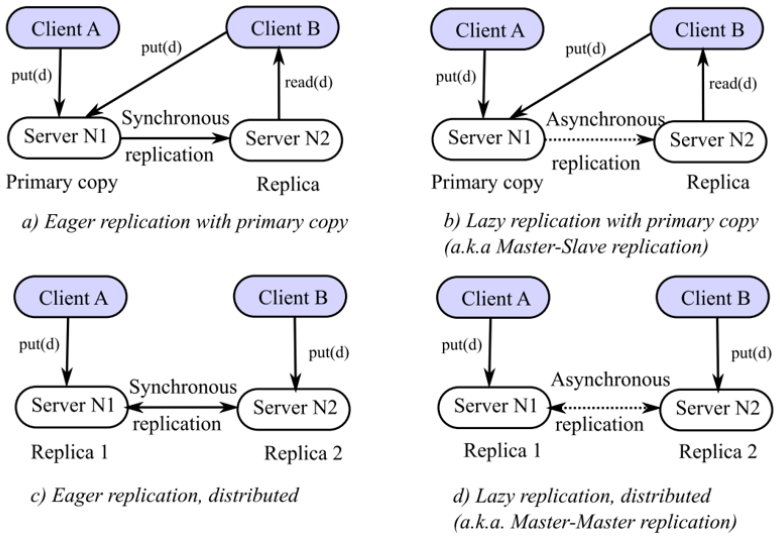


Figure 3.8: Replica synchronization alternatives, from [AMR+11]

a) A user can only modify the primary (or master) copy, and his/her changes are immediately propagated to any other existing copy (which can always be read by any user). Only after being properly propagated and changes acknowledged by all servers, the user receives confirmation.

b) A user can only modify the primary (or master) copy, and receives confirmation of this change immediately. His/her changes are eventually (i.e., sooner or later) propagated to any other existing copy (which can always be read by any user).

c) A user can modify any replica, and his/her changes are immediately propagated to any other existing copy (which can always be read by any user). Only after being properly propagated and changes acknowledged by all servers, the user receives confirmation.

d) A user can modify any replica, and receives confirmation of this change immediately. His/her changes are eventually (i.e., sooner or later) propagated to any other existing copy (which can always be read by any user).

LHS (i.e. eager replication) corresponds to the traditional (and strong) concept of consistency, while RHS (i.e., lazy replication) corresponds to the more innovative (and weaker) concept of eventual consistency. The behaviour can be fine tuned as will be discussed in Section 3.1.4.2.

3.1.4.1 Eventual consistency

Strong consistency is not compatible with highly distributed data management that among other things entails relaxing ACID (Atomicity, Consistency, Isolation, Durability) properties. As enunciated by the CAP theorem, distributed NOSQL systems must relax them as well as the traditional concept of transaction to cope with large-scale distributed processing. As a result, data consistency may be compromised but this enables the creation of fault-tolerant systems able to parallelize complex and time-consuming data processing tasks.

Thus, the concept of lazy replication gives rise to eventual[15] consistency, which is a concept weaker than the traditional one found in RDBMS (as in ACID transactions). It does not mean the database becomes inconsistent in the presence of updates, but it can temporarily be. Thus, it can be inconsistent for some time, but it will always tend to be consistent. More formally, it can be stated as "if the database stops receiving updates, it will eventually converge to a consistent state". Obviously, a database in production would never stop receiving updates, so its consistency is a sisyphean[16] task.

[15] "Eventual" means happening or existing at the end of a process or period of time (be aware, in Spanish or Catalan it is a false friend).
[16] `https://en.wikipedia.org/wiki/Sisyphus`

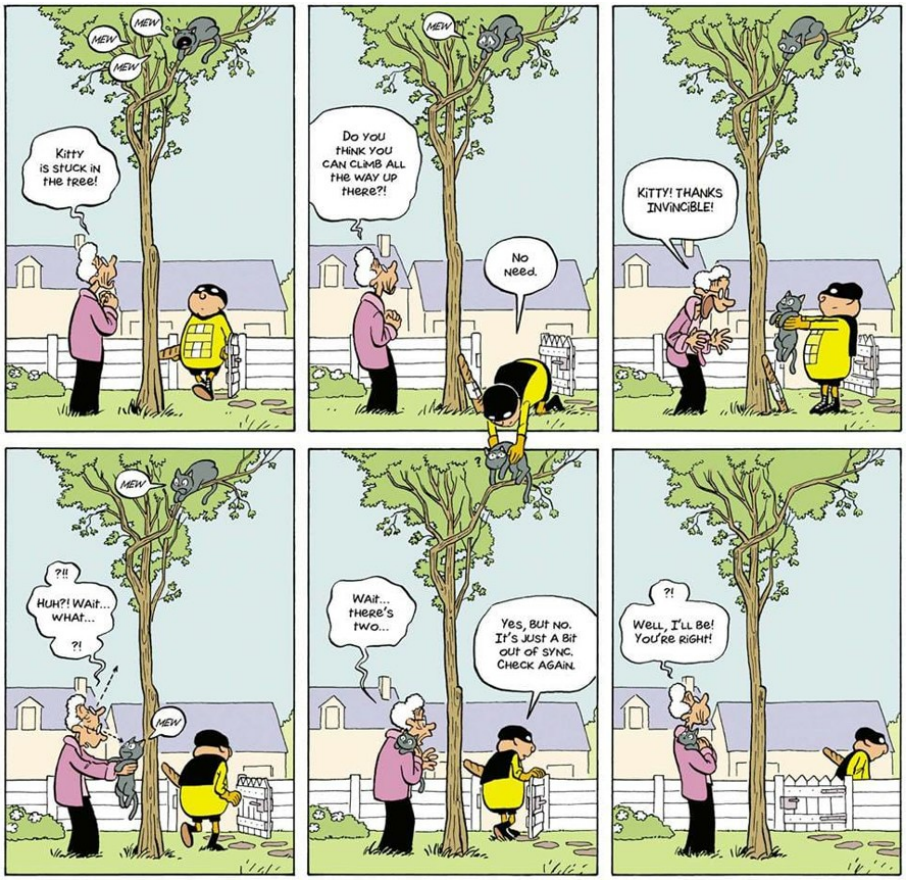


Figure 3.9: Eventually consistent stripe, by Justin Travis Waith-Mair

This idea is nicely illustrated in Figure 3.9.[17] Surprisingly, in the third and fourth vignettes there are two cats, but it is eventually solved and in the sixth one there is again only one (as it should have always been).

Therefore, in Cloud databases, where we cannot have strong consistency without affecting availability, the choice should be either one or another depending on the application. For example, in finances, better to opt for a less performant but safer strong consistency; but in social networks, one may sacrifice some consistency to avoid unnecessary locks and improve user experience. This does not mean at all that in the latter case the system in always unstable or inconsistent. As soon as there is not any network cut, the system might guarantee consistency of replicas. Then, on detecting a cut, the system must choose entering one of two modes. The first one simply disables all operations and waits for the network to be restored. In the second mode, the system keeps on working, even if this generates inconsistences, that must be solved later. Once the network is restored, a process known as partition recovery starts and tries to compensate mistakes. In some cases, logs obtained from both sides of the network cut can be merged and replayed to obtain a consistent result. If this still generates some conflicts, some other mechanisms can be implemented like majority voting or simply last writer wins.

3.1.4.2 Replication Management Configurations

To look at the different possibilities of eventual consistency, we need to define three different parameters:

N Number of replicas

W Number of replicas that have to be written before commit

R Number of replicas that need to be read before commit

[17] `https://medium.com/the-non-traditional-developer/dealing-with-promises-in-javascript-99d94105a972`

In systems like MongoDB[18], the values of these parameters can be freely chosen (see Section 6.3.3), giving rise to different concepts and situations. We can firstly define the concept of "Inconsistency window" as the time during which $W < N$. Notice that in this case, copies not written yet contain only the old data. Since all copies (either updated or not) are available and freely chosen, a user could read any of them. Thus, if the client reads only one of those not yet updated, it will retrieve obsolete/inconsistent data. To mitigate that, the client could read more than one copy. If both sets of machines (i.e., R and W) overlap, the user would detect the problem and keep on reading until getting the right answer (notice that since clocks in different machines are independent, the only way to identify the right data is majority voting). The more copies are read, the more unlikely is that none of them was updated. Therefore, the higher R, the more likely it is to get consistent results, but the slower the system will be. If we achieve $W + R > N$, we have strong consistency, because it is impossible that after checking R copies, none of them is right. On the contrary, $R + W \leq N$ corresponds to eventually consistent configurations, because all our R reads could still correspond to obsolete copies hence returning inconsistent results to the user.

On the other hand, $W < \frac{N+1}{2}$ implies that two write sets could be disjoint and hence independently confirmed to the corresponding users. This will generate an update conflict resulting in the loss of one of those updates (even if already confirmed to the user). In general, the lower W, the faster and more likely the written, but the higher the risk of conflict and consequently data loss. On the other hand, the lower R, the faster the reads and more likely to read inconsistent data. On the contrary, higher values for either W or R will difficult successfully finishing operations (depending on the probability of network cut).

Some typical configurations are:

Fault tolerant system $N = 3; R = 2; W = 2$

Massive replication for read scaling $N \gg 1; R = 1$

Rear One-Write All (ROWA) $R = 1; W = N$

3.1.5 Challenge IV: Query Processing

Given a query, producing an access (a.k.a. execution) plan, results to be a massive task, since the number of possible translations to be considered is simply out of consideration (note that, among many other parameters, it depends on physical structures, algorithms implementing the operators, order between operators, data statistics and the system configuration). Finding the optimum is computationally hard (NP-complex in the number of involved datasets), potentially resulting in higher costs than just retrieving the data. Therefore, query optimizers apply heuristics to work under certain assumptions that make this task feasible. Consequently, a DBMS does not necessarily find the optimal access plan, but just obtains an approximation, in a reasonable time. Produced access plans tend (in most cases, at least) to be close enough to the optimum but in any case, the optimizer does not even guarantee that it will be an affordable access plan.

The new main actor that comes into play in DDBMS is obviously the network (and the communication overhead it entails). Thus, distributed query optimization works over the same principles upon which centralized query optimization is based, but extended to consider data locality and availability of processing resources.

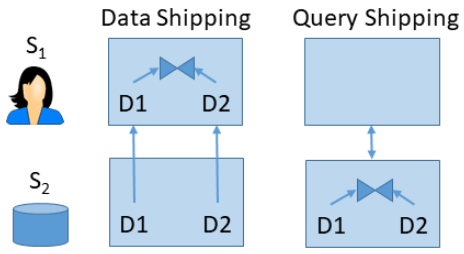


Figure 3.10: Data Shipping vs. Query Shipping

Shipping data through the network is, in general, not efficient and distributed query optimizers try to minimize the amount of data shipped. To simplify the explanation, let us assume that we only have two sites, one containing the data and another one where the user issues the query. In general, data shipping can be considered in any

[18] R and W correspond to parameters "readconcern" and "writeconcern", respectively.

operation, but we are going to focus now on joins because they are more relevant given their complexity. An exemplifying scenario would be that sketched in Figure 3.10, which contains two sites (namely, S_1 and S_2). Suppose now that a query containing $D_1 \bowtie D_2$ is issued in S_1 but these two datasets (neither fragmented nor replicated) are stored in S_2. At this point, we must decide where to execute the join.[19] Note that this is a crucial issue, because needed data must be shipped to the chosen site, which might involve a fair amount of data. For this reason, a new operator is introduced in the process tree, sometimes called "exchange", which explicits data being shipped from one site to another. Then, we have two possibilities:

a) With **Data shipping**, the data is retrieved from the stored site and sent to the one executing the query. This avoids bottlenecks on frequently used data, as the former only provides data but does not execute any operation (i.e., LHS in Figure 3.10). It is clearly a good option if the result of the operation is larger than its inputs.

b) Using **Query shipping**, the evaluation of the query is delegated to the nodes where data is stored. Oppositely, this avoids transferring inputs (i.e., RHS in Figure 3.10) and results specially useful when these are larger than the result.

c) To add an extra level of complexity, there is a **Hybrid** solution by means of semi-join reductions. This strategy has been widely accepted in distributed environments and has been proved to be of great value to decrease the amount of data sent over the network. A semi-join is a regular join operation, with the only difference that the output contains those attributes of one input dataset (either left or right). Formally, being A_i the attributes of R, the left semi-join between R and S is $R \ltimes S = \pi_{A_i}(R \bowtie S)$. Symmetrically, being B_i the attributes of S, the right semi-join between R and S is $R \rtimes S = \pi_{B_i}(R \bowtie S)$. The benefit of such approach is that it facilitates combining data and query shipping in the same operation. Specifically, it reduces the communication overhead, because only join attributes are firstly sent (from S_1 to S_2, and then only those tuples known to be in the result are eventually sent back (from S_2 to S_1). In fact, when sending tuples in the second step, the DDBMS does not even need to send the whole tuples, but just projections of them containing those attributes needed to answer the current query. As a negative aspect, note that we are performing more operations. Concretely, to decide whether it is worth or not to use a semi-join strategy, we should check, at least, that the size of the dataset to be shipped (D_1) is bigger than the projection of the join attributes to be sent from S_1 to S_2 plus the result of the semi-join ($D_1 \ltimes D_2$). However, the needed statistics to make such decision might not be available. As general rule, a semi-join should be considered if we have a small join selectivity factor (i.e., the result of the operation is a small percentage of the input dataset).

In general, none of the three options prevails over the others and the best solution is a mixed shipping strategy (as we can see in MongoDB or Spark), which trades the cost of transferring data against the overload generated in a frequently-queried site if data is not transferred. But again, the complexity problem pops up, since the amount of different alternatives to analyze, dependant on the size of the inputs and outputs of every operation in the process tree, is simply unaffordable. To make the problem even more complex, query optimizers in DDBMS do not only have to consider the network traffic, but also maximize the benefit as much as possible from parallelism as well as the existence of replicas. Consequently, given the inherent complexity of the problem, they tackle all this by introducing new heuristics and assumptions to simplify the search space, which generate new sources of potential mistakes.

3.1.5.1 Phases of query optimization

As already discusses, query optimization is an NP-hard problem with an immense search space that is tackled by means of heuristics and dynamic rules depending on the query pattern, which tend to get close to the optimum solution in most cases. Thus, it is important to know how the optimizer at hand works (i.e., which rules and heuristics it uses) to detect deviations from the optimal performance and correct them, when possible (e.g., adding or removing some indexes or fragments). This is the basis of what is known as physical tuning, which consists on deploying the most appropriate data structures (e.g., creating indexes, fragments, and updating statistics) and parameters to produce the best access plan.

All in all, a distributed optimizer must go through some extra stages with regard to a centralized one. Consider Figure 3.11, which sketches a typical query optimization process in a DDBMS. We can see that in a distributed

[19] We only consider here binary joins. The problem is more difficult for multi-way joins that operate at the same time more than two datasets.

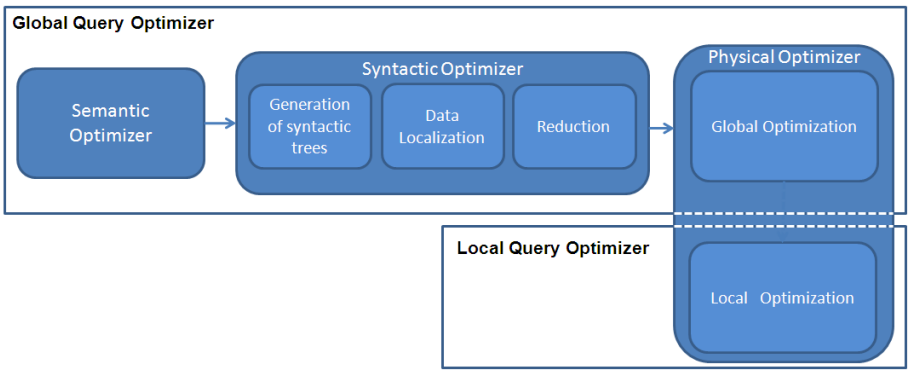


Figure 3.11: Optimization process in a DDBMS

query optimizer not only syntactic optimization phase must be extended, but also the physical query optimizer has more work to do, which is split between global (happening at the coordinator) and local (happening at workers).

Semantic optimizer performs some basic checks in terms of attribute names and types, and tries to optimize the query just rewriting it into an equivalent, but more efficient one (e.g., in the case of a RDBMS, it transforms SQL into SQL without any language translation). Since it plays the same role as in a centralized architecture, we will not deepen anymore in this component.

Syntactic optimizer rewrites the global query (which was issued in terms of the global schema) into a set of fragment query (according to data location) and identify tautological fragments which can be removed (known as reduction). It starts by representing the input query in an algebraic form (namely a syntactic tree). In the leaves of this tree we initially find the datasets, which are then replaced by the corresponding reconstruction operation depending on the kind of fragmentation applied (i.e., if horizontally fragmented the dataset is reconstructed by means of unions, and if vertically fragmented by means of joins). To do so, this stage requires the fragmentation schema available in the global catalog (see Figure 3.5). Then, alternative trees are generated by means of equivalence rules and heuristics, basically reordering joins (see Section 3.1.5.2) and pushing down projections and selections. After restructuring the trees, the reduction phase aims at determining those subtrees that will produce empty relationships (i.e., no data coming from that subtree is needed, because the fragment does not contain anything relevant to the query). To do so, fragment expressions are analyzed regarding the query predicates and projections to find contradictions.

Physical optimizer splits into two different stages: Global (adding communication, exchange operators) and Local (choosing the appropriate structures as well as their access paths, which maps to traditional physical optimization).

Global optimizer Performed at the coordinator site where the query was submitted, it is responsible for generating alternative process trees that are expected to minimize network use and benefit from parallelism as much as possible. We can see it as a cost-based layer, which does not consider access paths nor physical structures (i.e., disk accesses, which will be later carried out by the local optimizer), but exclusively focuses on communication costs and parallelism. Thus, it decides at which site each operation is going to be executed and inserts communication primitives (a.k.a. "exchange" operators) into the access plan. To know which communication operators must be added, we need to know availability of data and how those at different sites are related. To this end, replication plays a major role, and it requires the allocation schema available in the global catalog (See Figure 3.5).

Local optimizer Instantiated once at each worker site, each of them uses the information in the corresponding local catalog to decide which indexes to use, which is the most promising algorithm, etc. This exactly coincides with the tasks performed by a centralized DBMS, and therefore, we will not deepen anymore in this component.

Sometimes (specially if we look at concrete DBMS implementations), the line between syntactic and physical optimization is blurred. As a simple way to distinguish both tasks, we can say that syntactic optimization aims at

finding the best operation order in the tree, typically based on rules and without considering physical structures nor algorithms; while physical optimization is expected to be cost-based and decides the concrete algorithm to be used in each operator (depending on physical structures available and their access paths, communication costs, algorithms available, etc.)

3.1.5.2 Kinds of process trees

In this section, we focus on which alternative sequences of operators (a.k.a. syntactic trees) we have to execute each query. We focus on joins, as most DDBMS do, this being the most expensive operation and source of complexity. Thus, the syntactic optimizer deals with the problem of how to represent join trees and how to model and seek through the search space to find the optimum execution order.

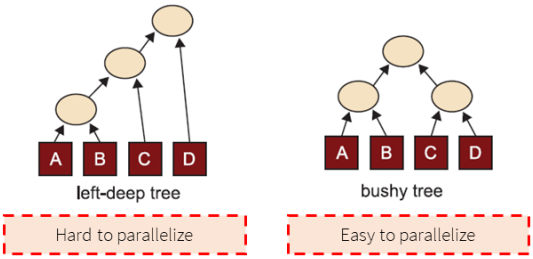


Figure 3.12: Kinds of process trees

Thus, DBMS in general, restrict the search to few specific topologies of trees. Bushy trees, sketched at RHS of Figure 3.12, allow operators to have either original datasets or intermediate results in both inputs, and consequently generate a large search space, hence, they are overlooked by centralized DBMS. Relevantly, note that left/right-deep trees (also known as linear trees), at LHS of Figure 3.12 are trees such that restrict all its operators to have at least one orinal dataset (i.e., not an intermediate result obtained from a previous operator) as input and therefore, the query optimizer only needs decide about the order of joins (i.e., much smaller search space). In practice, there is no deal in choosing between a right-deep or left-deep tree. Nevertheless, left-deep trees have been traditionally used to represent join trees for the sake of visualization. In a left-deep tree, the intermediate result produced by previous joins is always in the left-side of the operator (which is by convention assumed to be the outer loop of the row/block nested loop join algorithm). For example, consider the LHS in Figure 3.12. There, four datasets `A`, `B`, `C` and `D` join each other (the concrete join algorithm is not relevant now). In this example, two datasets are first joined (in this case, `A` and `B`). Next, the result obtained from this is joined to `C`, which, in turn, is joined to `D`. Note, however, that even after restricting the topology of the tree, still the number of linear trees we can generate in general is factorial (specifically, $\Theta(N!)$, where N is the number of datasets to join).

Nevertheless, distributed environments are not that fond of linear trees, because they limit the kind of parallelism we can apply. As we will see in Section 3.1.5.4.2, bushy trees offer some benefits in terms of parallelism, because operators in different branches can be executed completely in parallel without waiting one for the finalization of the other. Indeed, in our example at RHS of Figure 3.12, $\texttt{A} \bowtie \texttt{B}$, and $\texttt{C} \bowtie \texttt{D}$ are absolutely independent and can be executed completely in parallel. It is simply for this reason that distributed query optimizers should not limit their search to linear trees. Bushy trees, however, generate many more execution alternatives than linear trees as, now, it is not mandatory that one of the inputs is one of the original datasets.

Typically, the search space to find all possible linear trees is built by means of dynamic programming, which generates all possible plans by applying the associativity and commutativity properties of joins, breadth-first, in a deterministic way. In some cases, the time to compute all the alternatives for bushy trees could be even higher than the actual execution time of the query with a suboptimal tree. Therefore, optimizers are not that exhaustive when building bushy trees as they are on building linear trees. In this case, they use heuristics (such as focus on branches with projections and selections first) to bound the search space. Usually, the syntactic optimizer chooses between built trees based on the size of intermediate results, which usually give a final access plan not far away from the optimum. Thus, during the search among the different syntactic trees, join tree $\texttt{J}_1$ is decided to be more efficient than $\texttt{J}_2$ if the total size of intermediate results of $\texttt{J}_1$ is expected to be smaller than that of $\texttt{J}_2$. Obviously, updated statistics about the selectivity factor of joins, table cardinalities and attribute statistics are needed to properly estimate such cost (which is not always trivial).

Next, for each alternative tree generated (importantly, each of the alternatives found should be considered separately) the optimizer (based on well known equivalence rules between Relational expressions) simply pushes down selections and projections as close to the leaves (i.e., original datasets) as possible, which does not generate more alternative trees.

There is still another big challenge when dealing with distributed environments, because also different allocations of the syntactic tree operators to sites could be generated. Indeed, the problem is even bigger, since the DDBMS should decide first how many sites are really necessary in the execution of each query. So, since considering all possible alternatives in the immense search space is clearly unaffordable, as it already was in the simplest case of centralized optimizers, it is usually assumed that all available processing sites are used for every query. Thus, under the assumption that all available site can be used and trying to maximize data locality, the physical optimizer generates different assignments of operators to sites and hence introduces data exchange operators in the given syntactic tree.

3.1.5.3 Cost models

As a result of this assignment to sites, we will obtain a set of alternative process trees (representing, each one, an access plan). We must decide now which of these process trees is the most efficient one and, to do so, we need a cost model that quantifies how expensive each strategy is. Such decision is a hard problem with lots of parameters to be considered: the communication cost through the network, processing capabilities of each site, cardinalities of intermediate results generated, etc. Furthermore, note that other non-functional requirements (generally known as constraints) like data freshness, or site availability could also make the problem more complex. As result, most current solutions just simplify the problem to consider exclusively the size of intermediate results and communication costs derived from them. Thus, solutions provided in these scenarios are also cost-based, like in centralized databases, but the dominant factor is usually the bandwidth usage instead of disk accesses. Nevertheless, some networks are rather fast and sending data over the network can be even more efficient than performing disk accesses. Also, even distribution of data is assumed or, commonly, communication cost between sites is assumed to be constant. This might not be true for WANs and hence lead to skewed results, but it drastically simplifies the problem.

It is important to be aware of the system settings and assumptions made to decide which kind of cost estimation suits it better. In general, the cost function of the query optimizer refers to machine resources such as disk space, disk input/output, buffer space, CPU time, and network bandwidth. In a distributed environment, having all these statistics up-to-date is harder than in a centralized DBMS. Fistly, gathering statistics from data stored in all sites (i.e., statistics about fragments and replicas) is not cheap, and the global catalog being replicated or stored in a single site can also affect their management. Irrespectively of how hard it is to get them, the cost of an access plan can be estimated as the sum of local costs (provided by the local optimizer) plus global cost.

- Local processing:
  - Average CPU time to process an instance (T_{CPU})
  - Number of instances processed (#inst)
  - I/O time per operation ($T_{I/O}$)
  - Number of I/O operations (#I/Os)
- Global processing:
  - Time to issue a message (T_{Msg})
  - Number of messages issued (#msgs)
  - Transfer time (to send a byte from one location to another) (T_{TR})
  - Number of bytes transferred (#bytes but it could also be expressed in terms of packets)

The first set of items measures the local processing time. Mainly, the local cost has to take into account that of central unit processing (i.e., number of clock cycles needed), as well as that of disk I/O (i.e., number of transfer operation to/from disk). The second set of items measures the global processing time (i.e., communication costs). Thus, it considers channel occupation to initiate / send messages (e.g., for synchronization purposes between sites) and bandwidth spent sending data.

Once we have all those, different access plans of a query can be compared with respect to either total resources used or latency (a.k.a. response time). Be aware of the consequences of each approach: total resources used does not consider parallelism, whereas latency of a single query does not consider the overall system throughput. To obtain overall resources, we can assign weights/prices (W_x) to each component, the total cost (usually unitless) could be, accordingly, expressed as:

$$W_{CPU} \cdot T_{CPU} \cdot \#inst + W_{I/O} \cdot T_{I/O} \cdot \#I/Os + W_{Msg} \cdot T_{Msg} \cdot \#msgs + W_{TR} \cdot T_{TR} \cdot \#bytes$$

All in all, the knowledge we require to compute this cost, as in the centralized case, is basically related to the size of data and statistics needed to obtain the selectivity factor of operations and hence estimate the size of partial results generated by intermediate operators in the process tree. The main problem with this approach it that it does not account for the use of parallelism. Alternatively, to accommodate parallelism in our formulas, we can use a cost model expressed in terms of latency, and actually measuring response time (and not only resources needed to execute the query). For example:

$$T_{CPU} * seq_{\#insts} + T_{I/O} * seq_{\#I/Os} + T_{msg} * seq_{\#msgs} + T_{TR} * seq_{\#bytes}$$

Where $seq_{\#x}$ is the number of x (being x either instances, I/O operations, messages or bytes) which must be done sequentially (i.e., non-parallelizable) for the execution of the query. The first problem we would find then, is how to estimate what is and what is not parallelizable. Still, if we had that, the concurrent use of the four different resources is not considered in that equation. To do so, under the assumption that none of them waits for any of the others, we should actually take the maximum of all four components instead of the sum (see the effect of synchronization barriers in Section 3.1.5.4). It should be obvious that estimating latencies is much harder than purely resources, since the former needs to account for all the many different kinds of parallelism and how they impact the response time.

3.1.5.4 Kinds of parallelism

In a DDBMS it is important to employ parallel hardware effectively. Essentially, parallelism is obtained by breaking up the problem in small logic pieces and processing them in different processors (note that in the context of DDBs, these processors reside in different sites). Thus, serial algorithms need to be adapted to multi-thread environments, where the input dataset is divided into disjoint subsets (a.k.a. fragments) to be processed in independent parallel tasks. On the one hand, it is expected to reduce the response time (at least, when dealing with very large databases) and provide scalability and high-availability. On the other hand, this raises a potential problem about concurrency and contention conflicts when accessing resources and, furthermore, it may negatively impact the overall execution time (i.e., throughput) if we add the processor time consumed at all sites to coordinate and synchronize resources.

In general, we can study the behaviour of Big Data applications in the Cloud by using the Bulk Synchronous Parallel (BSP) model of concurrent computation (see [Val90]), which distinguishes three components:

1. **Computation** represents the tasks we want to perform.

2. **Communication** corresponds to the messages exchanged between tasks to coordinate their activities.

3. **Synchronization barriers** are the points where different tasks are required to provide their results before the processing can continue (e.g., both inputs of a join operator in a data flow are necessary before performing it, hence, such operator would generate one of these barriers).

Figure 3.13 illustrates the problem generated in this kind of computation, which is based on sequential waves of parallel tasks, and the difficulty to estimate the duration of jobs from purely the duration of the individual tasks. At LHS, we can see the ideal scenario where all the tasks (i.e., arrows) in the same wave take exactly the same time and finish at once, not generating any delay at all at synchronization barriers (i.e., red bars). Unfortunatelly, reality is better represented by RHS, where every task has a different duration and all those in the same wave have to wait for the slowest one to finish. Thus, the execution time of a wave is not the minimum or the average time of tasks, but rather the maximum. Obviously, this means that the more skewed the distribution of execution times of tasks is, the more delay we will accumulate in every synchronization barrier.

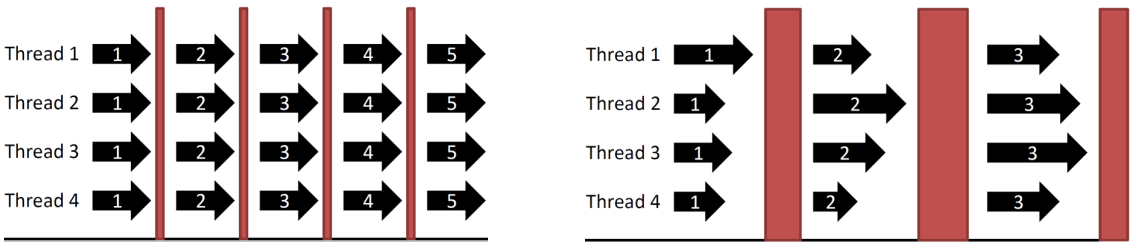


Figure 3.13: Bulk Synchronous Parallel Model, from [HCC+13]

Always following BSP model, there are still different alternatives for a DDBMS to benefit from parallelism. Namely, inter-query parallelism and intra-query parallelism. We talk about inter-query parallelism when several queries are executed in parallel (i.e., tasks correspond to queries and synchronization is only that imposed by contention). However, this is easily achieved by any DBMSs supporting concurrency, not necessarily a distributed one. Thus, in this section we are focusing on intra-query parallelism, where different parts of the same query are executed in parallel.

Lack of parallel programming skills are known to be a problem in many software engineering projects, but fortunately, algebraic Relational operators naturally benefit from parallelism even in the presence of fragmentation and/or replication. Thus, no specific parallel algorithms are needed and we can keep the discussion at the Relational level. Hence, for intra-query parallelism, we have, two alternatives:

Intra-operator Several parts of the same node in the process tree are executed in parallel (i.e., tasks correspond to processing different fragments of data and synchronization barriers to the reconstruction of the result). For example, a selection over a fragmented Relation (e.g., R, which has fragments; R_1 and R_2) can be executed in parallel by selecting over different fragments at each site and eventually merging the result obtained from each of them.

Inter-operator Several nodes of the same process tree are executed in parallel (i.e., tasks correspond to algebraic operators and synchronization barriers to confluences of edges in the syntactic tree). For example, two selections over two different Relations (e.g., R and S) that reside in different sites, can be executed in parallel, benefiting from distribution.

3.1.5.4.1 Intra-operator parallelism Intra-operator parallelism is always based on fragmenting data. Since query processing manipulates sets of tuples, these sets can be split into disjoint subsets, so that each result is generated only once and eventually merged. In case of *a priori* fragmentation, we can benefit from it. However, even if the dataset has not been fragmented before the query execution, the DDBMS can fragment it on-the-fly to benefit from intra-operator parallelism.

Indeed, the input(s) of an operation can be dynamically (i.e., at query execution time) fragmented and parallelized. Thus, no fragmentation is required beforehand. Specifically, there are different options for this kind of dynamic fragmentation.

a) Random (or round-robin) balances the load but blinds the searches.

b) Range facilitates directed searches, but needs accurate quartile information.

c) Hash allows directed searches, but performance dramatically depends on the hash function we choose, which will always have problems (i.e., skewness) in the presence of attributes with few distinct values.

In case of dynamic fragmentation, a new property has to be added to each operator in the process tree. This property must contain information about the fragmentation strategy (i.e., random, hash, etc.) being used, the specific fragmentation predicates and the number of fragments produced (e.g., see Section 8.2.3.2).

For example, if we want to join R and S, we can fragment S into S_1 and S_2 and now, by replicating R in two different sites and distributing both fragments of S, we can parallelize it as $R \bowtie S_1$ and $R \bowtie S_2$. Even more, if R is also fragmented by means of the corresponding join attribute (i.e., we obtain R_1 and R_2), we do not need to replicate anything, but simply ship half dataset to the corresponding site, because $R \bowtie S = (R_1 \bowtie S_1) \cup (R_2 \bowtie$

S_2). Thus, we not only gain parallelism, but also efficiency from the point of view that we expect those two smaller joins to be cheaper than a single join of bigger datasets.

In general, for binary operators in the process tree (such as join or set operators) the fragments can be generated dynamically. Moreover, only the attributes available for that operator are considered as candidates to generate the fragmentation predicates. For example, consider the following scenario:

- `R(rid, a1, a2)`,
- `S(sid, rrid, b1, b2)`, where `rrid` is a foreign key to `R(rid)`,
- and the following process tree: $(\pi_{rid,a_1}(R)) \bowtie_{rid=rrid} (\pi_{rrid,b_2}(S))$.

In this case, if we want to parallelize the join (note it is the only binary node in the process tree), the DDBMS can dynamically horizontally fragment its inputs and distribute them. Typically, the fragmentation strategies applied by DDBMS are hash or random strategies, which can be easily applied without human intervention (i.e., using a predefined hash function or randomly distributing tuples to the available sites). However, note that, in case of using hash, the DDBMS must pick an attribute to be used in the fragment predicates. In our example, there are only two candidates for each join input: `rid` or $\texttt{a}_1$ for the first one (note that $\texttt{a}_2$ cannot be used), and `rrid` and $\texttt{b}_2$ (i.e., `sid` and $\texttt{b}_1$ cannot be used) for the second one. If the database had two different sites, the DDBMS might decide to use a hash function (for example, in case of numbers, checking if they are even or odd) over `rid` and `rrid` and accordingly, distribute both inputs. Even values would be sent to one site, and odd ones to the other.

$$\sigma_{rid \bmod 2=0}(\pi_{rid,a_1}(R)) \bowtie_{rid=rrid} \sigma_{rrid \bmod 2=0}(\pi_{rrid,b_2}(S)) \cup \sigma_{rid \bmod 2=1}(\pi_{rid,a_1}(R)) \bowtie_{rid=rrid} \sigma_{rrid \bmod 2=1}(\pi_{rrid,b_2}(S))$$

Now, the DDBMS can parallelize the join, because matching values from both inputs will be sitting in the same site and never in the other.

About unary operators in the process tree (e.g., projection or selection), they can benefit from intra-operator parallelism only in the presence of static (i.e., pre-existing) fragmentation (in the sense that the fragmentation strategy must have been provided in advance; i.e., at design time). A typical strategy to fragment datasets and benefit from parallel unary operators is to store data according to small ranges of values among the available storage sites (even if the produced fragments do not really make sense from a conceptual point of view, like ranges of social security numbers).

Consider the example just introduced above and suppose now that the $\texttt{a}_1$ is a number representing the age of people, and we also know that this attribute is frequently queried in our database. In this case, we could decide to fragment `R` with regard to $\texttt{a}_1$. Two possible alternatives to be considered are to either manually define the ranges (e.g., $1-18$, $19-30$, $31-45$, $46-65$ and $66-99$), or use a hash function (e.g., simply using the modulo operation). The tuple will be stored in one site or another depending on the result of the defined function. For example, if five sites are available, each of the five ranges would be placed in a different site (if hash alternative were chosen, a modulo five function would have a similar effect).

In this way, queries accessing a single value (not worth to be parallelized) will be executed in just one site but with much less data, while queries involving many values of $\texttt{a}_1$ spread over several sites are executed in parallel balancing the load among the different sites. For example, queries containing selections such as $\texttt{a}_1 = \texttt{x}$ where `x` is a number between 1 and 99 will be processed in just one site, while queries involving large ranges, such as $\texttt{a}_1 <> \texttt{x}$ would be executed in parallel, as it implies to check all the tuples. In the latter, the DDBMS will take advantage of all five sites to parallelize such operation and eventually unite the results produced at each site.

As an extension to this strategy, compound fragmentation strategies are also possible. For example, one column can be hashed to decide the site where to store the tuple and a different column to decide the storage device (a.k.a. disk) inside the site. However, note that using the same column and hash function at both levels is a bad idea, since tuples assigned to the same site are guaranteed to produce the same hash value (and thus, all of them would be assigned to the same device).

3.1.5.4.2 Inter-operator parallelism Two different possibilities are available for inter-operator parallelism, depending on the topology of the process tree (see Figure 3.12). In the easy case, if it is a bushy tree (note that, in this section, we talk about bushy trees in general, either join trees or trees involving any other operation), independent branches can be obviously executed in parallel. Alternatively, in presence of linear trees (either right-deep or left-deep), parallelism is more tricky, as different strategies must be applied depending on the operator (in some cases, it might not even be worth). For example, nodes can still work in parallel if tuples are pipelined from one operator to the next one (i.e., one site can process one tuple while another site or processor is processing another).

As explained in [Pit18], pipelining can be implemented demand-driven or producer-driven. Demand-driven pipelining is typically used in centralized DBMS, whose query executor basically creates a chain of nested iterators (a.k.a. cursors), having one of them per operator in the process tree. Thus, the system simply pulls from the root iterator, which transitively propagates the call through all other iterators in the pipe. In principle, this does not allow parallelism, but it can be easily implemented by adding a buffer to each iterator. This way, they would eagerly generate next rows without waiting for a parent call. Thus, the producer operator leaves its result in an intermediate buffer and then the consumer one takes its content asynchronously. It is important to notice that the producer can generate output tuples only until the buffer becomes full.

The demand-driven approach with buffers is absolutely equivalent to producer-driven pipelines, the only difference being in the trigger of the process (either root or leaves). Both require buffers and both naturally parallelize the processing. Nevertheless, by using buffers, we risk stalling the system. As explained in [Joh09], stalls happen when an operator becomes ready and no new input is available in its input buffer. Note that this stalling scenario is propagated like a bubble through the buffers chain (i.e., if the input buffer of one operator becomes empty, it will stop generating tuples and its output buffer will soon become empty, affecting next operations in the chain). Similarly, if a buffer become full, its operator will stop processing. Consequently, we can conclude that in the end the pipeline progresses at the pace of the slowest operator. This can be alleviated by the size of buffers, but not completely solved.

		Occupancy	Latency
Serial system		T	T
Parallel	No stalls	T/N	$h \cdot T/N$
	Stalls	$T/N + k$	$h \cdot (T/N + k)$

Table 3.4: Comparison of latency and occupancy in a stalling system

To better visualize the problem, let us define *latency* as the time to process the query, and *occupancy* as the time until which the DBMS can accept more work. In other words, latency tells us the overall amount of time needed to *answer* the query (i.e., from when the query is issued until the query is answered completely), whereas occupancy tells us the time needed to *execute* one operator (i.e., the time needed by every component/processor to execute every single query piece). Assuming that we have N operators to parallelize, h is the height of the tree (either a linear or bushy tree), T time units are required for the whole query and k is the delay imposed by communication overhead and imbalance in case of stall, Table 3.4 shows the values of those two measures.

In the case of a non-parallel system, both latency and occupancy coincide, since only one query is processed at a time. Then, if we assume a uniform distribution of the query workload among operators, each of them would take T/N time to execute. Thus, the system can start accepting a new query once the first operator is done (i.e., after T/N). However, the execution of the current query will need to progress through h levels in the tree (each with several operators running in parallel, but taking the T/N time each). Nevertheless, this is not realistic, because it does not consider overheads, imbalances or synchronization barriers. Therefore, to reflect that, we should consider an added cost k to every operator.

Importantly, note the benefit of having bushy trees instead of linear trees (see Figure 3.12). For linear trees, in these formulas, h always equals N, but this is not true in bushy trees. In both trees have the same amount of operators to parallelize (i.e., $N = 3$). However, while the left-deep tree has height 3 (note we do not count leaves), the bushy tree has only height 2. Remember that, by definition, each operation described in a linear tree always takes as input exactly one dataset (no more). Consequently, the height is always equal to the overall number of operations to parallelize. In general, the larger number of operations to parallelize, the bigger is the difference in height between a linear tree and its bushy counterpart. Consequently, although bushy trees can also suffer from stalling, they are not only easier to parallelize, but in the general case, they better exploit parallelism.

3.2 Measures for parallelism

In parallel query processing, there is a trade-off between latency (a.k.a. response time) and throughput (i.e., the rate of production, here in terms of queries over a time unit). We can reduce the latency of a query by parallelizing it. To achieve this, we need to partition the database, or certain tables, into subsets that can be processed in parallel. This requires also to adapt the existing algorithms (e.g., a full table scan) into their parallel counterparts (e.g., fragment-based scan), roughly by introducing some extra operations to synchronize the different processors being involved (the more processors, the more latency reduction, but also the more synchronization operations required). It is obvious that such synchronization operations also consume resources that could otherwise be devoted to other queries or users. Thus, serially executing queries maximizes the overall throughput of the system for a fixed amount of resources (since it eliminates the synchronization mechanisms required by parallelism). However, in this situation some queries would take too much time to yield an answer. Nevertheless, this loss of performance can be compensated by adding extra hardware. The question is, then, to what extent can latency and throughput really be improved by adding hardware resources. In other words, to what extent can the scalability of a system be quantified for a given set of resources. For this, we distinguish two metrics:

Speed-up, which measures the performance gain when new hardware is added but the problem size remains constant.

Scale-up, which measures the performance gain when the problem size is altered at the same time that new hardware is added.

Ideally, we should get linear (close-to-linear) scalability in terms of both scale-up and speed-up. Precisely, linear speed-up states that N processing units solve in T/N time units a problem that takes T time units to be solved by a single processing unit. Linear scale-up states that if a single processing unit takes T time units to solve a problem of size S, then it also takes T time units to solve a problem of size $N \cdot S$ by N processing units. In the context of DDB that hold vasts amounts of data, we are specially interested in achieving linear scale-up. This is, however, hardly achievable due to the communication costs incurred in any distributed systems. We, next, present two formulas to quantify the degree of improvement that a system can achieve introducing more processing units (i.e., Amdahl's law and the Universal Scalability Law).

3.2.1 Amdahl's law

This law, which takes the name of its author Gene Amdahl, measures the theoretical speed-up that a system can achieve by introducing parallelism. The principle behind it is that parallel computing with many processors is useful only for highly parallelizable programs. To this end, Amdahl's law measures the maximum improvement that a system can achieve by taking into account the fraction p of code that can be parallelized, and the number N of processing units.

$$S(p, N) = \frac{1}{(1-p) + \frac{p}{N}}$$

The takeout of this formula is that, we as system developers, must aim to minimize the $(1-p)$ serial time, representing the piece of work that will be always executed serially regardless of the number of processing units. For example, if a program needs 20 hours to complete using a single thread, but a one-hour portion of the program cannot be parallelized, therefore only the remaining 19 hours ($p = 0.95$) of execution time can be parallelized, then regardless of how many threads are devoted to a parallelized execution of this program, the minimum execution time cannot be less than one hour. Hence, the theoretical speed-up is limited to at most 20 times the performance of a single thread[20].

The behavior of Amdahl's law is depicted in Fig. 3.14. We can observe that if $p = 1$ (i.e., unrealistically 100% of the code is completely parallelizable), we achieve linear scalability, yet as this value decreases the theoretical speed-up radically decreases.

[20] `https://en.wikipedia.org/wiki/Amdahl's_law`

		p		
		100%	50%	25%
N	1	1.00	1.00	1.00
	2	2.00	1.33	1.14
	3	3.00	1.50	1.20
	4	4.00	1.60	1.23

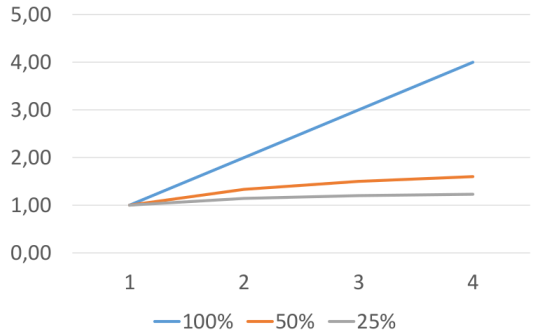


Figure 3.14: Evolution of $S(p, N)$ for different values of p and N

3.2.2 Universal Scalability Law

The Universal Scalablity Law (USL) builds from the observation that Amdahl's law does not take into account the cost of communication and syncronization among parallel processors[21]. Considering the case of shared nothing architectures, where all kinds of communication occur over the network, this is not a negligible cost. Proposed in [Gun07], the USL aims to express that there is precisely a moment where adding extra parallel processing units worsens the overall performance.

The USL, which takes the notion of *universal* as it works for both software and hardware, measures the system's capacity improvement (i.e., the improvement in performance). It takes as argument the number N of processing units, like Amdahl's law, which can be number of threads for software or number of CPUs for hardware. Then, the formula builds on two case-based constants, namely the system's contention (σ) and the system's consistency delay (κ). The former (i.e., σ), is the counterpart of p in Amdahl's law, and thus models the fraction of the algorithm that is not parallelizable (i.e., must be serially executed). The latter (i.e., κ), models how much the different parallel units require communication.

$$C(N) = \frac{N}{1 + \sigma \cdot (N-1) + \kappa \cdot N \cdot (N-1)}$$

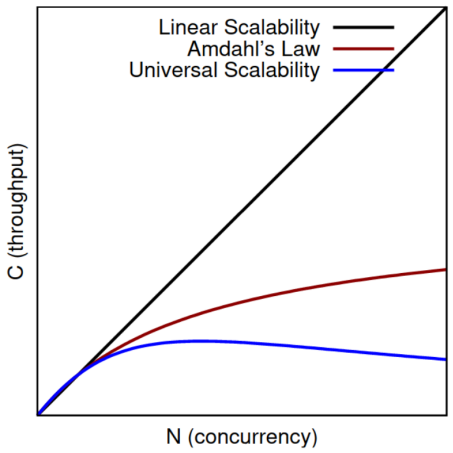


Figure 3.15: Behavior of different measures for parallelism, by Baron Schwartz and Ewen Fortune

Figure 3.15 depicts the general behavior of the USL with respect to Amdhal's law and linear scalability. Note that, when we do not account for the system's consistency delay (i.e., $\kappa = 0$) then the USL is equivalent to Amdahl's law. Linear scalability is achieved when we do not account for both the system's contention and consistency

[21] https://wso2.com/blog/research/scalability-modeling-using-universal-scalability-law

delay (i.e., $\sigma = 0$ and $\kappa = 0$). However, at this point, it is crucial to realize that from some point on, adding more machines worsens performance. This reflects reality and can be appreciated in that the denominator grows quadratically in N, while the numerator only grows linearly in N.

3.2.2.1 A method to determine the optimal scalability parameters

In this subsection, we present a method to determine the optimal number of parallel processors N such that $C(N)$ is maximized[22]. The method assumes that the system to be evaluated is available, and that it is possible to evaluate its throughput on different values of N.

i) The first step requires evaluating the system under different values of N. This will yield a set of throughput values $C_1, \ldots, C_n$. For the different values of N, we also compute $x = N - 1$ and $y = \dfrac{N}{C(N)} - 1$, being $C(N) = \frac{C_N}{C_1}$.

ii) Next, using least-squares regression, we can fit a second-order polynomial of the form $y = ax^2 + bx + 0$. Higher R^2 values (e.g., 99%) tell us that the curve fits well the measured data points.

iii) From the obtained polynomial coefficients a and b, we can derive the parameters σ and κ. The relationships between the coefficients and the parameters are as follows:

$$\sigma = b - a$$
$$\kappa = a$$

iv) Finally, we can plot the USL formula instantiated with those parameters, and find the maximum value of $C(N)$ to obtain the corresponding N. Such N is the optimal system's size in terms of computing nodes that maximizes the throughput.

[22] `https://www.percona.com/sites/default/files/white-paper-forecasting-mysql-scalability.pdf`

Chapter 4

Distributed File System

Managing unprecedented large amounts of data requires distributed and parallel processing. However, we have not presented yet what are the challenges for the storage of these data. This is, how should we manage the low-level aspects at the file system level (i.e., disk blocks), so that features such as fault-tolerance or synchronization are transparent to developers.

Due to the nature of a distributed file systems (DFS), some of the classical choices in the design of centralized file systems (FS) are no longer valid. When distribution is in the Cloud, problems can only be amplified. Precisely, the new generation of large-scale DFS needs to be designed to deal with and satisfy the following requirements:

Efficiently file management DFS are meant to store data generated by applications such as system logs, social network interactions or container formats (e.g., video formats). Thus, it is not rare that the size of a single file is in the range of GBs to TBs, and we expect a large number of those. It is assumed that small files (i.e., KBs or few MBs) should rarely exist in those cases, and there is no need to optimize the system to deal with them.

Efficient append Most files are updated by appending new data rather than overwriting existing ones. Small writes at random positions can also be supported but it is considered that they do not need to be optimized.

Multi-client Allow for multiple clients concurrently appending data to the same file. Minimal syncronization overhead must be generated.

Optimal sequential scans Reading workloads consist of two kind of access patterns: sequential scan and random. In sequential scans, also referred as *large streaming reads*, a range of the file is scanned usually spanning several MBs. In random accesses, also referred as *small random reads*, the system reads few KBs at some random offset of the file. Mainly the former is considered relevant in this case.

Resilience failure Component failures are the norm rather than the exception. Machines composing the DFS are usually inexpensive and their characteristics are far from the ones found in high performance computing systems. In such a setting, where the DFS can be composed of hundreds or thousands of this kind of machines, it is very common that hardware failures occur frequently. Thus, this kind of situations must be monitored and detected, and when they happen there must be recovery mechanisms in place to revert potential inconsistencies.

Capacity	Nodes	Clients	Files
10 PB	10,000	100,000	100,000,000

Table 4.1: Capacity of a GFS cluster

It is clear then, that traditional FS do not cope well with the abovementioned assumptions and new solutions are required. In this chapter, we will present the Google File System (GFS), the first large-scale DFS that later became Apache Hadoop Distributed File System (HDFS), the most wide-spread open source DFS today. Table 4.1 shows the magnitudes that a GFS cluster is expected to manage.

4.1 File structure

GFS is the internal DFS at Google, whose open source version is Apache Hadoop Distributed File System (HDFS), which was implemented from the details and results presented in [GGL03]. GFS was designed to handle very large collections of unstructured or semi-structured data.

Figure 4.1: The Apache Hadoop logo, with its characteristic elephant

As we will see in Section 4.3.1, there exist addons to natively manage some tabular structure in the data, but the only original structure of a file is simply a division into blocks. Indeed, when a new file is to be written in the file system, it is just first split into blocks of a given size (64MB by default in GFS, but configurable).

Technology	**Coordinator**	**Worker**
GFS	Master	ChunkServer
HDFS	NameNode	DataNode
HBase	HMaster	RegionServer
MapReduce	JobTracker	TaskTracker
Spark	Driver	Worker
MongoDB	ConfigServer	Shard

Table 4.2: Node names for every technology

4.2 GFS architecture

There do not exist any essential differences to the backbone implementation of HDFS with respect to GFS. The only important aspects to be aware of are naming and notation differences or configuration default values. Both these technologies follow a Coordinator-Worker architecture (wrongly named Master-Slave in the past). The coordinator node is responsible for tracking the available state of the cluster and it basically manages the worker nodes, which are those doing the actual work (Table 4.2 shows the different nomenclature used for different technologies).

Next, we present some of the design decisions in GFS:

Files are split in chunks. This is the minimal unit of distribution. Regardless of the amount a small file takes, the system will always allocate a chunk for it. In GFS the default chunk size at the moment of publication was 64MBs, a value that can be customized per file.

Chunks can be replicated. By default, there will exist three replicas of each chunk, although this value can also be customized. To guarantee robustness, each replica must be stored in a different chunkserver. If the cluster cannot replicate a chunk for the required value (e.g., because there are less available chunkservers than the replication factor), then we say that the chunk is *underreplicated*.

In-memory namespace. Serving thousands of client applications, must entail a minimal overhead. Thus, in order to reduce lookup cost, the data structure containing the file hierarchy and references to their chunks resides in memory of the coordinator node. Precisely, when a GFS cluster is started, part of the startup time is devoted to poll chunkservers in order to get their chunks and recreate the file namespace structure.

Bi-directional communication. The coordinator and chunkservers have mechanisms to send both data and control messages among them. Chunkservers can also send control messages to each other, such as commit requests and ack confirmations, although in most cases they will be exchanging chunks of data.

Single point of failure. A GFS cluster is coordinated by a single node. This creates a single point of failure (SPOF) situation, such that if the coordinator fails the whole cluster becomes unavailable. To overcome this situation, it is common to maintain a failover replica (a.k.a. mirror).

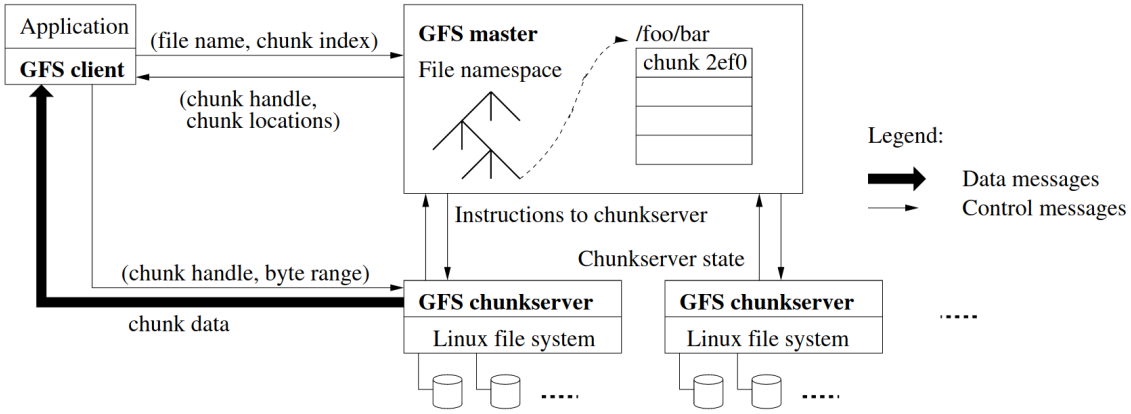


Figure 4.2: GFS architecture, from [GGL03]

Thus, as depicted in Figure 4.2, a cluster is composed by a coordinator (called *master* in GFS) and many workers (called *chunkservers*). Hence, the architecture of a GFS cluster is governed by a single machine in charge of accepting connections from external applications, and instruct them on where to fetch data from, which is achieved thanks to the *file namespace* (or *directory tree*), that contains the hierarchy of directories and files, as well as references to the chunkservers storing the data of each file. The single coordinator keeps all the file system directory in memory, which takes approximately 1GB per petabyte of data. Oppositely, a GFS cluster can be composed of hundreds or thousands of chunkservers, which are the machines in charge of storing data in disk and serving them directly to client applications.

The concept of chunk is also dropped in HDFS in favour of the already overloaded term *block*. It is important to take into account that one HDFS block is not equivalent to a physical disk block, the former being a virtual concept. A single HDFS block will span multiple physical blocks in disk, which usually take 8 KBs. Note that, the default block size in HDFS is doubled with respect to the original GFS chunk size, and is thus set to 128MBs. Given a file each of its chunks is stored in a potentially different chunkserver (randomly chosen) and replicated in multiple nodes. By default, every chunk is replicated three times, but this is configurable.

4.3 Data Management

At this point, it is important to remember that we are dealing with a file system and not really a fully functional DBMS.

4.3.1 Data Design

Next, we present the main features in HDFS such as writing and reading files, or guaranteeing fault tolerance.

4.3.1.1 Fragmentation

In this section, we discuss the different storage layouts available for tabular data and exemplify them with their corresponding instantiation for HDFS. The kind of storage layout that is used will directly impact how data are physically stored in the blocks, and thus how they will later be read. We distinguish three kinds of layouts: *horizontal*, *vertical* and *hybrid*. Each concrete layout is beneficial for a specific kind of workload. In the following subsections, we present the main characteristics of each, as well as the most representative file format implementing them.

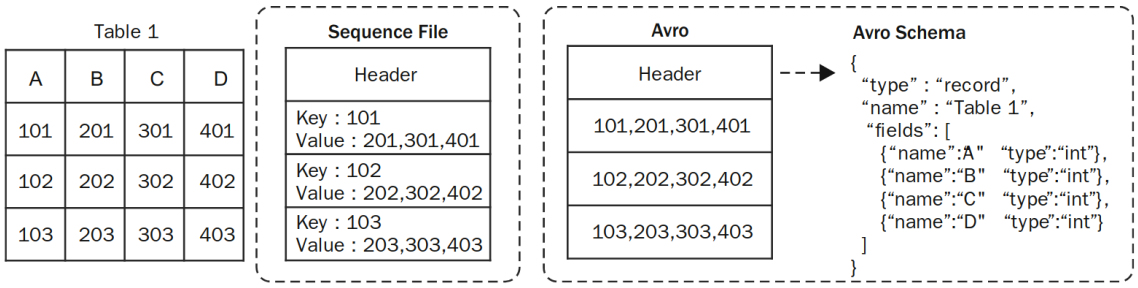


Figure 4.3: Example of horizontal layout (SequenceFile and Apache Avro formats), from [MAR+20]

Horizontal Layouts are organized row-wise, and the attributes of each row are stored together. For this reason, a horizontal layout especially suits scan-based workloads. However, if a query refers to a small subset of columns, then this layout results in a low effective read ratio, since non-required columns will be fetched anyway. In HDFS, the horizontal layout is implemented by SequenceFile[1] and Apache Avro[2] data formats. SequenceFile is a special type of horizontal layout storing binary key-value data, whereas Avro explicitly splits data into attributes inside every row. In other words, it embeds some basic schema information. Figure 4.3 shows an example of a table and its corresponding representation in SequenceFile and Avro.

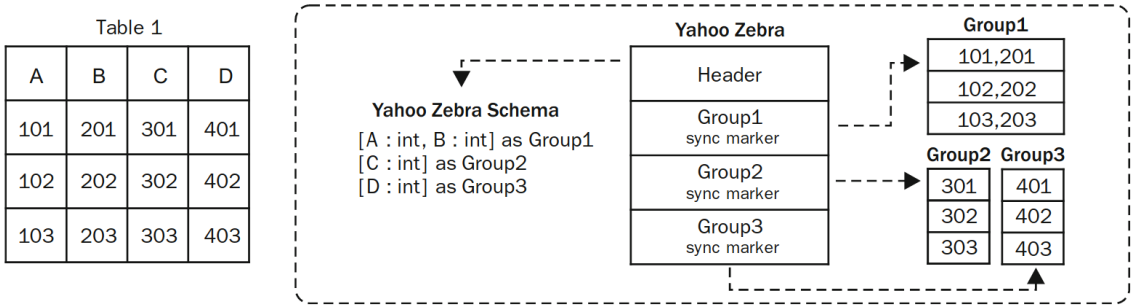


Figure 4.4: Example of vertical layout (Yahoo Zebra format), from [MAR+20]

Vertical Layouts divide each file into columns (oppositely to horizontal layouts) and store each of them separately, which is beneficial for workloads reading few columns. Thus, these layouts excel in projection-based workloads. Yahoo Zebra, illustrated in Figure 4.4, is an implementation of this kind of layout for HDFS. Zebra also allows to group columns together, but without any horizontal fragmentation.

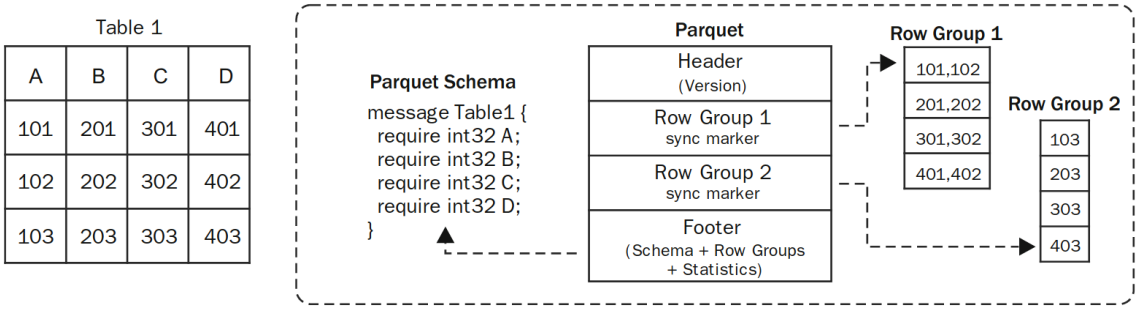


Figure 4.5: Example of hybrid layout (Apache Parquet format), from [MAR+20]

[1] https://wiki.apache.org/hadoop/SequenceFile
[2] https://avro.apache.org

Hybrid Layouts are a combination of horizontal and vertical layouts. There exist two alternative implementations: either the data is divided horizontally and then vertically, or vice versa. Both cases are especially helpful for combinations of projection and selection operations. There are many implementations of this kind, but the most popular ones in HDFS are Apache Optimized Row Columnar (ORC)[3] and Apache Parquet[4], both primarily fragmenting data horizontally and then vertically. Figure 4.5 exemplifies Parquet.

Features	Horizontal		Vertical	Hybrid	
	SequenceFile	Avro	Zebra	ORC	Parquet
Schema	No	Yes	Yes	Yes	Yes
Column Pruning	No	No	Yes	Yes	Yes
Predicate Pushdown	No	No	No	Yes	Yes
Metadata	No	No	No	Yes	Yes
Nested Records	No	No	Yes	Yes	Yes
Compression	Yes	Yes	Yes	Yes	Yes
Encoding	No	Yes	No	Yes	Yes

Table 4.3: Comparison of data formats

Comparison of features. We, finally, present a comparison of some characteristics of the different storage layouts and their representative file formats. Table 4.3 presents side by side the features available in each. It can be noted that all formats except SequenceFile, store some basic schema information. Storing the schema helps during the data serialization and de-serialization phases (from memory to disk and vice-versa) by avoiding the need to cast the data at the application level, a costly operation due to the impedance mismatch. We can also observe that both vertical and hybrid layouts provide support for column pruning, which means they only fetch the required columns and do not perform unnecessary disk reads. Hybrid layouts can also push down the selection predicates into the storage layer because they keep some basic statistical information (e.g., minimum and maximum value per attribute in each partition) that helps in filtering the records while reading. Additionally, vertical and hybrid layouts have support for nested records which helps in storing bag, map and custom user data types. It can also be noted that hybrid layouts have additional features but they also have a significant overhead when writing and therefore when reading the same amount of data (due to the amount of accompanying metadata). Nowadays vertical layouts have been surpassed by hybrid ones, as they support all their features.

4.3.1.1.1 Heuristic file format choice. We can conclude from the feature comparison that each format provides a different set of features that will affect the overall performance when retrieving the data from disk. Generally, hybrid layouts perform well if a subset of columns is read. Alternatively, horizontal layouts perform well if all, or most of the columns are read. The work presented in [MRA+16] gives a set of heuristic rules to decide the most suitable file format depending on the kind of query to be executed. This are summarized as follows:

- Use SequenceFile if the dataset has exactly two columns (i.e., a key and a value).
- Use Parquet if the first operation of the data flow aggregates data or projects some of the columns.
- Use Avro if the first operation of the data flow scans all columns, or performs other kinds of operations such as joins, distinct or sort.

4.3.1.1.2 Cost-Based file format choice. The cost-based model relies on a wide range of statistical information that is summarized in Table 4.4, containing system constants, data statistics, workload statistics as well as layout variables. The system constants are generally based on [JQD11]. We only extended the list with specific variables related to the selectivity factor and storage layouts. We assume the constants depending on the configuration of the environment (e.g., BW_{Disk}, BW_{Net}) are given and the statistics are collected during the data flow execution.

[3] `https://orc.apache.org`
[4] `http://parquet.apache.org`

Variable	Description
	System Constants
R	Replication factor
p	Probability of accessed replica being local
BW_{Disk}	Disk bandwidth
BW_{Net}	Network bandwidth
$Time_{Seek}$	Disk seek time
$Time_{Disk}$	Disk access time computed as $\frac{Chunk_{Size}}{BW_{Disk}}$
$Time_{Net}$	Network access time computed as $\frac{Chunk_{Size}}{BW_{Net}}$
	Layout Variables
RG	Row group of hybrid layouts
$Meta_{Layout}$	Metadata for a given layout
$Body_{Layout}$	Body of a layout
$Header_{Layout}$	Header of a layout
$Footer_{Layout}$	Footer of a layout
	Functions
$Size(X)$	Size of X in chunks
$UsedChunks(L)$	Number of chunks of layout L
$UsedRG(O)$	Number of row groups involved in operation O
$Seeks(Layout)$	Total number of seeks for a given layout
$Cost(O)$	Unitless cost of operation O
	Data Statistics
$\|F\|$	Number of rows in the file
$\|RG\|$	Number of rows of a row group
$Size(cell)$	Average cell (i.e., value) size
$Cols(F)$	Number of columns of file F
	Workload Statistics
$Proj(F)$	Number of columns projected from F
SF	Selectivity factor (i.e., percentage of rows) of an operation

Table 4.4: Parameters of the Cost Model

Moreover, it should be noted that we consider only I/O cost in our cost model, because it is the dominant factor in LAN.

$$Size(Layout) = Size(Header_{Layout}) + Size(Body_{Layout}) + Size(Footer_{Layout}) \quad (4.1)$$

$$UsedChunks(Layout) = \frac{Size(Layout)}{Size(Chunk)} \quad (4.2)$$

$$Seeks(Layout) = \lceil UsedChunks(Layout) \rceil \quad (4.3)$$

Independently of the kind of layout, the driving factor of our cost model is the file size. The body, together with the header and footer compose it (Equation 4.1). From that, we can obtain the number of chunks used (Equation 4.2) and the number of disk seeks we need to reach them (Equation 4.3). The number of seeks is equal to the total number of chunks rounded up, because one seek is required for every chunk, even if it is not full. Note that modern Solid State Disks (SSDs) also have seek time (i.e., time required to turn on the right circuit), however their seek time is much less (i.e., 0.1ms) compared to rotating disks (i.e., 4ms) [Kip15, LBKS16]. Thus, our cost model still applies to SSD and we would only need to update the system constants accordingly.

In the next subsections, we analyze the cost of data writes and reads, because they are the dominant factors in the overall execution time of data flows. The write cost model estimates the data volume footprint of each layout as well as the cost incurred in writing it, while the read cost estimates the cost of an operation depending on the access pattern. Regarding the latter, given the simplicity of a file system (far from that of a DBMS) only three operations are possible (namely full scan, projection, and selection).

$$W_{WriteTransfer} = \frac{Time_{Disk} + (R-1) \cdot Time_{Net}}{Time_{Seek} + Time_{Disk} + (R-1) \cdot Time_{Net}} \tag{4.4}$$

$$\begin{aligned} Cost(Write_{Layout}) = {} & UsedChunks(Layout) \cdot W_{WriteTransfer} \\ & + Seeks(Layout) \cdot (1 - W_{WriteTransfer}) \end{aligned} \tag{4.5}$$

Write cost. First of all, we have to take into consideration that distributed processing frameworks are using DFS to store data into multiple chunks. Thus, the number of chunks of a file is used to estimate the overall writing costs. Given that a chunk consists of multiple contiguous disk blocks and inside it, sequential read is guaranteed, assuming that the chunk size is smaller than a disk cylinder, the write cost can be simply computed as the size of chunks plus the seek cost to locate the position of each. Nevertheless, since our cost model is thought for distributed processing frameworks, we further need to consider the replication factor R, and therefore the network costs for writing R copies needs to be added. We assume that the replication procedure is sequential (as it is in HDFS) and the multiple copies are written one after another. Equation 4.4 gives the weight of transferring a chunk by considering the network and the disk write against the seek costs. Finally, Equation 4.5 shows the total write cost taking both transfer and seek (i.e., the complemtary of transfer) weights into account.

In the following, we present the estimation of size for each horizontal, vertical and hybrid layouts, so this can be used to estimate chunks and seeks to used in the writing cost.

Horizontal layouts store data row-wise into the body section. Oppositely, metadata containing information such as schema and version, is written into the header and footer sections. Nevertheless, in some implementations, additional metadata is also written in the body with every row, for example, metadata used to separate each row or each column (i.e., its size is not constant and depends on the number of columns).

$$Size(Body_{Horizontal}) = Size(metaBody) + |F| \cdot (Size(metaRow) + Size(dataRow)) \tag{4.6}$$

Equation 4.6 estimates the size of the body by multiplying the average row size and metadata (i.e., $Size(metaRow)$) by the total number of rows, plus other metadata (i.e., $Size(metaBody)$) we may find in the body section.

$$Size(dataCol) = |F| \cdot Size(cell) \tag{4.7}$$

$$Size(Body_{Vertical}) = Size(metaBody) + Cols(F) \cdot (Size(metaCol) + Size(dataCol)) \tag{4.8}$$

Vertical layouts store each column independently (i.e., values of a column, which share the same data type, are stored consecutively) using a separator (i.e., $Size(metaBody)$) of fixed size between columns. Equation 4.7 provides the estimation of the individual column size, which is used in Equation 4.8 to determine the overall size of the body by multiplying the size of one column by the total number of columns.

$$UsedRG(Hybrid) = \frac{Cols(F) \cdot |F| \cdot Size(cell)}{Size(RG) - Size(metaRG) - Cols(F) \cdot Size(metaCol)} \tag{4.9}$$

$$\begin{aligned} Size(Body_{Hybrid}) = {} & \lceil UsedRG(Hybrid) \rceil \cdot (Size(metaRG) + Cols(F) * Size(metaCol)) \\ & + Cols(F) \cdot |F| \cdot Size(cell) \end{aligned} \tag{4.10}$$

Hybrid layouts are a combination of horizontal and vertical layouts. They divide rows into horizontal partitions known as row groups and each row group is further divided into vertical partitions storing each column separately, and inserting metadata (i.e., $metaCol$) between them. Additionally, they also store metadata (i.e., $metaRG$) for every row group. Thus, the total size of the body depends on the number of row groups being used, which can be estimated as in Equation 4.9. Furthermore, Equation 4.10 obtains the size of the body by multiplying the number of row groups by the size of metadata in a row group and by adding the total size of original data. Notice that the metadata of a row group is stored irrespectively of it being completely full, so this must be rounded up.

Read cost. This section presents the read cost model for scan, projection and selection operations. All data flow operations in current massively distributed processing environments use a full scan access pattern on the DFS, except projection and selection operations that are specifically supported natively in some storage layouts. Thus, we consider them separately in the following.

$$Size(Scan_{Layout}) = Size(Layout) + (\lceil UsedChunks(Layout) \rceil \cdot Size(Meta_{Layout})) \tag{4.11}$$

Scan reads all stored data from the disk, irrespective of the layout being used. The metadata (such as schema, statistics, etc.) stored inside header or footer sections, is read separately in each task. The reason is that the distributed processing engines (such as Hadoop and Spark) create a separate process for each task with its own virtual machine. Its memory is not accessible to other tasks (i.e., shared nothing architecture) and hence, forces to read all metadata in each task separately, and consequently, increases the reading size. The number of tasks is equal to the number of used chunks. Equation 4.11 estimates the scan size (under the assumption that the split unit that decides the number of processing tasks is the chunk), which can be used further to estimate the scan cost.

The scan cost purely depends on the number of used chunks to be read. Assuming the block is the transfer unit between disk and memory, there are three factors impacting the cost: the average seek time needed to locate a disk block cylinder, the rotation time to move the disk head over the cylinder to reach the block, and the transfer time to bring data in the block from disk into memory. Nevertheless, despite every chunk consists of multiple blocks on disk, it should be noted that DFS typically guarantee that all disk blocks are contiguous within one disk cylinder, under the assumption that the chunk size does not go beyond the cylinder size. This is why we do not need to consider seek time for all the disk blocks. Instead, we only consider seek time once for every chunk. Also, it can be empirically confirmed that rotation time is negligible, because modern hardware and operating systems implement very effective pre-fetching techniques. Furthermore, the cost model is also applicable to SSDs by simply assigning the right system constants. For the rest, since the basic unit of the cost model is defined in terms of bytes, all the estimations will remain the same.

$$W_{ReadTransfer} = \frac{Time_{Disk} + (1-p) \cdot Time_{Net}}{Time_{Seek} + Time_{Disk} + (1-p) \cdot Time_{Net}} \tag{4.12}$$

$$UsedChunks(Scan_{Layout}) = \frac{Size(Scan_{Layout})}{Size(chunk)} \tag{4.13}$$

$$\begin{aligned} Cost(Scan_{Layout}) &= UsedChunks(Scan_{Layout}) \cdot W_{ReadTransfer} \\ &\quad + Seeks(Layout) \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{4.14}$$

On the other hand, we have to take under consideration that in a distributed data processing framework data can be accessed remotely. Consequently, we introduce a probability p to indicate the likelihood of chunks being accessed locally (i.e., data shipping through the network is not needed to reach the operation executor). This is used to estimate the weight of transferring the chunk data compared to the corresponding seek time using Equation 4.12. Then, Equation 4.13 estimates the total number of chunks read and Equation 4.14 provides the scan cost taking both the seek and the transfer cost into account with the corresponding weights.

Projection helps in fetching only some columns from disk (skipping others) to save some I/Os, and is typical of Vertical layouts. Its cost depends on the support provided by each layout. Horizontal layouts do not provide specific support for projection operation, but actually use a full scan to bring all the data into memory and only afterwards discard the unnecessary columns. Therefore, its cost is exactly the same as that of scan (i.e., Equation 4.14). Despite being characteristic of Vertical layouts, Hybrid layouts also natively support projection and behave similarly. Thus, we will focus on the latter because of being more popular than purely Vertical layouts.

$$UsedRows(RG) = \frac{|F|}{UsedRG(Hybrid)} \tag{4.15}$$

$$Size(projCols) = Proj(F) \cdot UsedRows(RG) \cdot Size(cell) \tag{4.16}$$

$$\begin{aligned} Size(Project_{Hybrid}) &= Size(Header_{Hybrid}) + Size(Footer_{Hybrid}) \\ &\quad + \lceil UsedRG(Hybrid) \rceil \cdot (Size(metaRG) + proj(F) \cdot Size(metaCol)) \\ &\quad + UsedRG(Hybrid) \cdot Size(projCols) \end{aligned} \tag{4.17}$$

$$\begin{aligned} Cost(Project_{Hybrid}) &= UsedChunks(Project_{Hybrid}) \cdot W_{ReadTransfer} \\ &\quad + Seeks_{Hybrid} \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{4.18}$$

First of all, we have to calculate the size of fetched data to estimate the cost. However, hybrid layouts store data into multiple row groups. Therefore, we first need the row group size to estimate the projection size. As each row group contains a subset of rows, we estimate it as in Equation 4.15. Furthermore, Equation 4.16 gives the size of the columns used in the operation inside a group, which is then used in Equation 4.17 to estimate the overall projection size. Similar to the scan cost, hybrid layout also reads metadata separately for projection in each task, which we consider in the projection size. Hybrid layouts also have a seek cost to be considered, which depends on the number of row groups needed given the overall size of the file (not only the result of the projection). Similar to previous cases, we can estimate the projection cost of hybrid layouts by appropriately weighting the transfer and seek times as in Equation 4.18.

Selection helps in fetching only some rows from disk (skipping others) to save some I/Os. As for projection, its cost depends on the support provided by each layout. Horizontal and Vertical layouts do not natively support this operation. They perform a full scan to bring all the data into memory and then filter them out based on the given predicate. Thus, their selection cost is the same as that of scan.

$$P(RGSelected) = 1 - (1 - SF)^{UsedRows(RG)} \tag{4.19}$$

$$\begin{aligned} Size(RowsSelected) &= \left\lceil \frac{SF \cdot |F|}{UsedRows(RG)} \right\rceil (Size(metaRG) + Cols(F) \cdot Size(metaCol)) \\ &\quad + SF \cdot |F| \cdot Cols(F) \cdot Size(cell) \end{aligned} \tag{4.20}$$

$$UsedRG(Select_{Hybrid}) = \begin{cases} UsedRG(Hybrid) \\ \cdot P(RGSelected) & \text{if Unsorted} \\ \left\lceil \frac{Size(RowsSelected)}{Size(RG)} \right\rceil & \text{if Sorted} \end{cases} \tag{4.21}$$

$$\begin{aligned} Size(Select_{Hybrid}) &= Size(Header_{Hybrid}) + Size(Footer_{Hybrid}) \\ &\quad + (UsedRG(Select_{Hybrid}) \cdot Size(RG) \end{aligned} \tag{4.22}$$

$$\begin{aligned} Cost(Select_{Hybrid}) &= UsedChunks(Select_{Hybrid}) \cdot W_{ReadTransfer} \\ &\quad + Seeks(Select_{Hybrid}) \cdot (1 - W_{ReadTransfer}) \end{aligned} \tag{4.23}$$

Oppositely, Hybrid layouts keep statistical information about data values in every column for every row group (typically, inside the header or footer sections), which helps in skipping some of the row groups that do not satisfy the predicate. Thus, the number of row groups to be read depends on the filtering condition and the sorting order of the column on which the selection is applied.

For unsorted columns, we can use the probability as in Equation 4.19 (borrowed from bitmap indexes [Car75]) to estimate the likelihood of any data in a row group satisfying the condition (i.e., a row group being fetched).

In Equation 4.21, this probability is used to obtain the expected number of retrieved row groups. However, if a column is sorted, then we are using the *Selectivity Factor (SF)* to estimate how much data is going to be read using Equation 4.20, which is later used in Equation 4.21 to calculate the fetched row groups for sorted columns (notice that all data fulfilling the condition is stored together if they are sorted on that column). Having the number of selected row groups, Equation 4.22 determines the size of a selection by adding up the total size of fetched row groups, metadata, header, and footer sections. As previously discussed about multiple reads of metadata in each task, we also consider this factor in the estimation of selection size.

Finally, this selection size can be used to estimate the total number of chunks and seeks as in Equations 4.2 and 4.3, which are then weighted as in Equation 4.23 to estimate the total selection cost.

4.3.1.2 Allocation

Random and massive distribution and replication of data, which are the two strongest points of GFS, allow to satisfy fault tolerance and high throughput access. Thus, it is the task of the coordinator to randomly assign the different chunks (which is the distribution unit) to servers. A random distribution is purely a mean to aim at a balance in the workload.

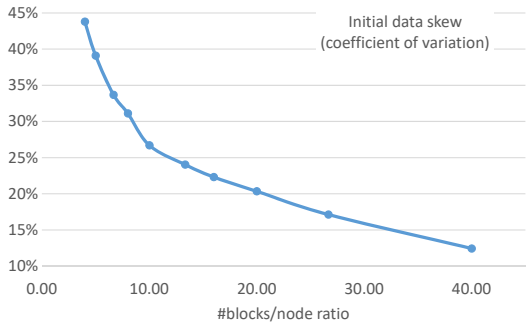


Figure 4.6: Change in the coefficient of variation depending on the number of blocks per node

Balancing allows GFS to have a great performance working with large datasets, since any read/write operation exploits the parallelism of the Cloud. However, at this point, it is important to understand the scale of GFS. Looking at the size of chunks, one should realize that we are dealing with very large files, but maybe it is not clear how large they should be. Considering that the processing relies on parallelism and that this is maximised for a uniform distribution of workload, it is clear that the more uniformly distributed the data are, the better. Figure 4.6 presents the results of an experiment in a chart corresponding to the coefficient of variation of the distribution of the number of HDFS blocks, depending on the average number of blocks (i.e., chunks) per node. These results show that having more blocks per node in the cluster increases the probability that the blocks will be better balanced over the nodes. However, it can also be seen that even for 40 blocks per node we still have more than 10% in the coefficient of variation, which is far from a balanced distribution. Considering that this experiment was for only eight nodes, it means that with a file of $8 \cdot 40 \cdot 64MB = 10,280MB$, its data was still far from being equally distributed in the cluster (which should clearly be our goal).

To correct a very likely skewed distribution of blocks in an HDFS cluster, since v2.7, Hadoop offers a stable version of an admin tool, called Balancer[5]. This examines the current cluster load distribution and based on the given threshold parameter (10% by default), it redistributes blocks across cluster nodes to achieve better load balancing. The threshold parameter denotes the percentage deviation of the load of each node, from the clusters average load, i.e., the coefficient of variation in Figure 4.6. Exceeding this threshold in either way (higher or lower) would mean that the node will be rebalanced. Besides periodically fixing the distribution of an HDFS cluster in use, Balancer is also useful when changing cluster's topology (i.e., adding new, initially empty nodes).

[5] `https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html#balancer`

4.3.1.3 Replication

One way to alleviate potential problems and skewness in the distribution of data is replication, but it may also boost performance by choosing the closest replica and reducing communication costs. However, replication also implies high availability, since different replicas can be used in case any of them becomes temporally unavailable. By default, GFS maintains 3 replicas of each chunk in different workers. The replication factor is a value that can be customized depending on the application requirements, which can be set globally for the whole system (with property `dfs.replication`) or at the level of file (with API call `setReplication`). If it is not possible to maintain the replication factor, the system informs the user about the specific chunks that are *underreplicated*. Detection of failures and tolerance is also managed from the coordinator. Periodically, by default every 3 seconds, the coordinator receives *heartbeat* messages from chunkservers. If a chunkserver systematically fails to send heartbeats to the coordinator, here for a 60 seconds period, then the coordinator assumes that the chunkserver is unavailable and corrective actions must be taken. In this situation, the coordinator looks up the file namespace to find out what replicas were maintained in the lost chunkserver. Such missing replicas are fetched from the other chunkservers maintaining them, and copied to a new chunkserver to get the system back to a robust state.

Stale replicas can appear either because a worker was down and its blocks were not updated or due to a failure of a write operation, the replica blocks are then older (i.e., have different content) than the primary replica. Such replicas are known as *stale* replicas, and detecting them is crucial to maintain consistency across the cluster. To that end, the coordinator maintains block version numbers to distinguish up-to-date and stale replicas. When the master aims to append new data to a file, it updates the block version numbers associated to the file's blocks. This mechanism allows to detect inconsistencies and flag stale replicas, which are deleted by default.

4.3.2 Catalog Management

The catalog of HDFS in centralised, and basically keeps all the directory of files in memory. Thus, it is clearly a single point of failure and can easily become a bottleneck. Consequently, there is the possibility of mirroring it in a secondary node.

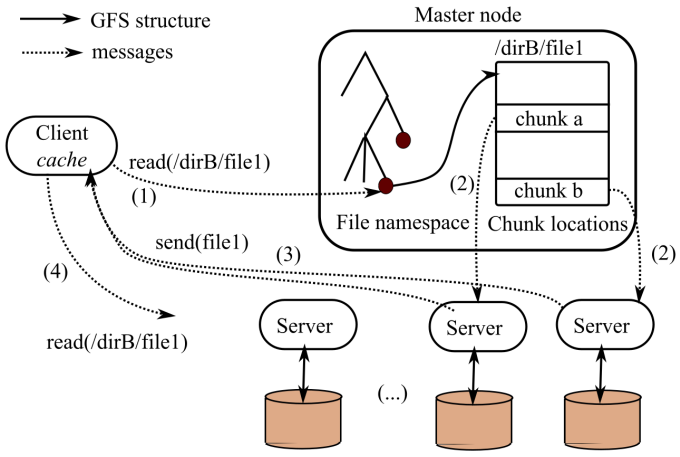


Figure 4.7: Reading a file in GFS, from [AMR+11]

However, there is a more elaborated mechanism that automatically releases the coordinator from some burden. First of all, we need to consider that many data flows are either iterative in nature (e.g., page rank), or simply rerun periodically (e.g., retraining). Thus, the same files are retrieved once and again. Reading a file entails sending chunks from chunkservers to the client application, which firstly requires the client application to get the corresponding list of chunks and locations. Thus, this list would be requested and served once and again, as well.

Figure 4.7 sketches the strategy to minimize the number of messages in the network, and specially the interaction with the coordinator in such scenario. The first time a file is requested, the client application cannot avoid (1) requesting the information from the coordinator. Then, the coordinator could send back to the client application the

list of chunks (and all replicas of each), but alternatively, in order to save one communication step, it (2) directly instructs chunkservers to (3) send the chunks composing the file to the client application. The coordinator chooses the replicas according to the closeness in the network to the client so that bandwidth is optimized.

Once the client application has read the file once, it keeps the locations of all the chuncks in a cache. Thus, in the likely case of requiring the same file in the near future, it does not need to disturb the coordinator again. Instead, it can (4) request directly the chunks to the corresponding chunkserver (notice that caching metadata is realtively cheap, requiring orders of magnitude less memory than caching the files themselves). Notice that we can somehow see the set of caches in all client applications as a strategy for the fragmentation and replication of the catalog.

Deleting files in HDFS is managed via the special directory `.Trash`. When a file is deleted, it is moved to such directory, and it is kept there in case it should be restored. After a parametrized amount of time a process is triggered to delete all files present in `.Trash`. This causes all blocks associated to the files to be freed. By default this process is automatically triggered at UTC 00:00. Alternatively, the user can trigger the process on-demand using the `expunge` shell command.

4.3.3 Transaction Management

Again, it is important to remember that we are talking about a simple file system. So, discussing ACID properties or talking about transactions does not make much sense in this context. Nevertheless, when it comes to synchronizing replicas, HDFS applies an eager/primary-copy strategy. Thus, writing can only happen on the primary-copy and its replicas are blocked until they are synchronized. Indeed, when a client application wants to write data to a file, it first communicates to the coordinator what is the target path and how many chunks it wants to write. For each chunk, the coordinator annotates how many replicas are to be stored (i.e., the replication factor), and where they will be stored. This information is sent back to the client application, who will write the first chunk, or *primary replica*, into a chunkserver (i.e., the *primary chunkserver*). In the background, chunkservers will talk to each other sending replicas of the chunk until a consistent state is reached (i.e., there exist as many replicas of the chunk as the replication factor). Finally, messages are sent back through the pipeline until the client application, who gets the confirmation of a successful write. If this message does not arrive, the client application can decide to retry the writing of the chunk again.

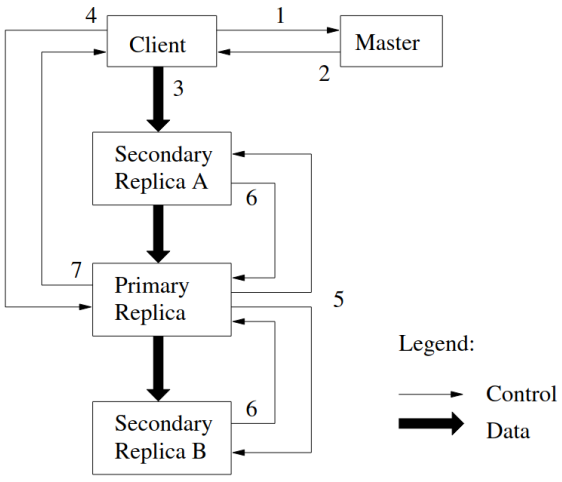


Figure 4.8: Management of primary replicas in GFS, from [GGL03]

The concrete workflow is summarized in Figure 4.8. Firstly, the client application (1) fetches from the coordinator (2) the list of replicas for the chunk to be updated. Then, the client application can (3) successively send data to all chunkservers (one by one in any order). Previously, the master would have granted a lease over the chunk to some of the replicas that will be considered primary. The request to (4) commit the changes must be sent to that primary replica, who will be in charge of (5) committing changes to all other replicas, (6) receiving confirmation

from them, and finally (7) acknowledging the changes back to the client application. Thus, we can see that file writes are actually pipelined and synchronously committed.

4.3.4 Query Processing

A file system is not a DBMS. Therefore, we should not be talking properly about queries, but purely reading and writing files. Indeed, we must assume append-only modification patterns (like in the case of logs), which is completely different from in-place updates in OLTP environments. Thus, writing relies on a simplified randomized distribution of data and reading on more sophisticated massive data processing environments like MapReduce and Spark (see Chapter 8).

Anyway, either the early MapReduce or the more sophisticated Spark still try to keep things as simple as possible and surprisingly rely on sequential access to be efficient. It is easy to read in many places about the importance of indexes and how they are fundamental to improve performance. However, even being true in many cases, they are not always the best option. It is clear that they go against simplicity, firstly because of the many kinds of structures that can be used, but most importantly because of the many candidates we have (roughly an exponential number of combinations of attributes). Moreover, we should never underestimate the power of sequential access.

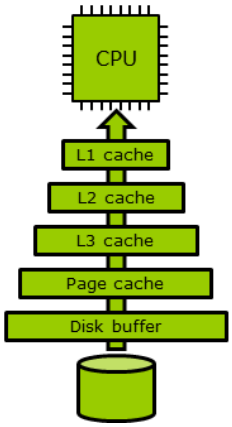


Figure 4.9: Sequential reading facilitates caching

Indeed, sequential reads heavily benefit from data locality, which is mostly ignored by random access. Let's assume a rotating disk, whose cost basically depends on three components: seek time (necessary to position the arm), rotation time (i.e., time waiting for the disk to spin until the head reaches the right sector) and data transfer (typically depending on the bandwidth).[6] Thus, the difference between random and sequential access is that the former does not find the data together (which result in a cost for n pieces of data of $n * (seek + rotation + transfer)$), while the latter does find all data consecutive (so, accessing the same n pieces of data becomes only $seek + rotation + n * transfer$). Moreover, sequential access pattern makes the next read absolutely predictable enabling pre-fetching,[7] and maximizes the effective read ratio by benefitting from the multiples layers of caching (as depicted in Figure 4.9). The closer to the disk, the more capacity memory has, but the higher its latency. The closer to the processor, the faster and smaller the memory is. Thus, finding the data in the top levels of the cache hierarchy is hard, but crucial to gain performance. Sequential access has a highly positive impact in the hit ratio of all caching system.

[6] You can assume seek time is approximately 12ms, rotation time 3ms and transferring 4KB of data takes 1ms.
[7] `https://en.wikipedia.org/wiki/Cache_prefetching`

Chapter 5

Key-Value Stores

Apache HBase[1] is an open-source, distributed, versioned, NOSQL store part of the Hadoop ecosystem. Designed following the specifications of Google BigTable as in [CDG+06], it leverages the distributed data storage of HDFS to provide random access capabilities. BigTable was created with the following characteristics:

Persistent: Data should be stored in disk.

Map: A data structure where some key helps to identify (i.e., retrieve) associated values.

Multi-dimensional: Each key does not identify a single value but potentially many.

Sparse: Most of the elements potentially identified by every key are unknown or inaplicable.

Sorted: In order to favour the retrieval of intervals of keys, these should be lexicographically sorted.

Distributed: The system should run in a cluster of machines facilitating parallelism.

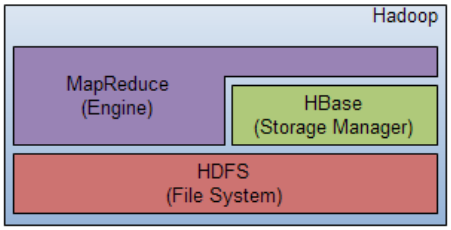


Figure 5.1: Logical architecture

The Hadoop ecosystem can be roughly sketched as a three level architecture in which we find HDFS (the file system) running at the lowest level, HBase (the storage manager) running on top of HDFS and finally MapReduce (the query execution engine) wrapping them so that data processing can be performed at both the file system and the database level (Figure 5.1 shows this logical architecture). Thus, HDFS physically stores the data in files; HBase manages the data, and provides basic indexing capabilities allowing random access; and finally, MapReduce is the query execution engine that filters, transforms and aggregates these data. As consequence, these technologies are relatively independent from each other in the sense that they do not form a single process running in a machine, but there is one independent process for each of them interacting to each other through the network. In a traditional setting a tuple is retrieved by querying the RDBMS, which forwards the message to the file system, which, in turn, retrieves the corresponding disk block and sends it back to the DBMS. Typically, these communication costs are disregarded since there is a tight coupling between the DBMS and the file system and such communication is performed in main memory. However, this is no longer true in an architecture like Hadoop since the file system (HDFS), the storage manager (HBase) and the query engine (MapReduce) do not form a single unit. On processing data, a message is forwarded to HDFS to retrieve the corresponding file block, which is then processed by HBase

[1] `https://hbase.apache.org`

to obtain the tuple and then sent to whomever requested it. Their communication is implemented via expensive network, which may be the "127.0.0.1" loopback interface in some cases.

5.1 Key-Value database model

Data are stored in HBase by following [key,value] structures. In such pairs, the key represents the row identifier and the value contains the row attributes. The [key,value] pairs are stored using the equivalent to well-known primary indexes for RDBMS, which physically sort rows on disk and build a tree-structure on top of it. A special characteristic to keep in mind is that data is physically sorted (a.k.a. clustered) by the key of the pair. Actually, HBase also supports horizontal fragmentation by these row keys. In other words, HBase splits each table by groups of consecutive keys, and stores the corresponding rows in different machines.

Such fragments are called "regions", which are the minimal distribution unit used by HBase. Data distribution is done according to the number of regions per node (i.e., RegionServers in HBase). Key-Value pairs are distributed depending on the region they belong to, but, in principle, regions are not guaranteed to be of the same size and hence data is not necessarily evenly distributed across the cluster. Additional features such as region splits and compactions were introduced to eventually achieve, in the presence of large enough data volumes, an even distribution among RegionServers.

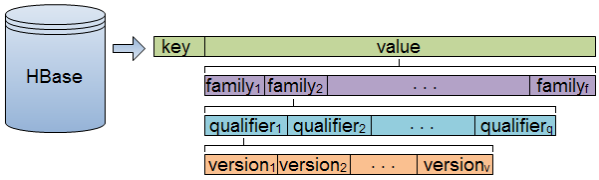


Figure 5.2: Internal structure of an HBase row

Moreover, HBase further structures the value to support vertical fragmentation (i.e., "families" of columns). Vertical fragmentation is relevant for read-only workloads since it improves the system performance, because non-relevant families (for the current query) are not read. Figure 5.2 sketches how a table row is stored in terms of families and qualifiers (a.k.a. column names). Note that qualifiers play a key role to decide which attributes must be stored together on disk by placing them in the same family. Families must be explicitly created by modifying the schema of the table. However, one qualifier belongs to a family and it is only declared at insertion time. Thus, providing enough flexibility as expected in a schemaless database. Note that a qualifier exists only inside one family, so two qualifiers are allowed to have the same name as long as they are from different families. Then, for each family and qualifier, there are versions (i.e., timestamps). Each combination of a family, qualifier and version determines an attribute value for a given key. Versioning keeps track of the n (configurable) most recent values of these attributes.

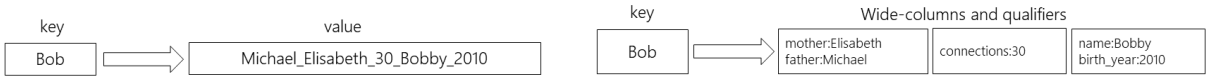


Figure 5.3: Example of Key-Value and Wide-Column storage

For instance, let's consider a table with data about people. At LHS of Figure 5.3, we have the representation of the row storing all the information of "`Bob`" separated by underscores inside a single value. Oppositely, at RHS of the same figure, we explicit that there are three kinds of information for each person (they might correspond, for example, to ancestors, social networks and personal data) that we store in different families. Thus, in the first of these families, we have his "`mother`" and "`father`", in the second the number of Facebook "`connections`" and in the last one his "`name`" and "`birth_year`". All these are qualifiers.

5.2 HBase Architecture

HBase offers a minimalistic functionality in terms of Put and Get operations, that respectivelly allow to store a [key,value] pair, and retrieve the value given the key. This is clearly far away from the complexity of a RDBMS

offering the whole standard SQL functionality. With a quick check you can compare that against the whole list of HBase shell commands[2] offering just few tens of basic functionalities.

Nevertheless, its strength is precisely in this simplicity, also reflected in its functional architecture, which is tightly coupled to HDFS. Indeed, data belonging to the same region must be stored in the same DataNode in HDFS (in order to avoid degrading performance). Otherwise, data would be unnecessarily spread all over the cluster regardless of vertical and horizontal partitioning strategies applied by HBase. Accordingly, there must be some communication between HDFS and HBase so data are stored where they are managed (*data locality principle*). This implies that a RegionServer must always run on top of one DataNode.

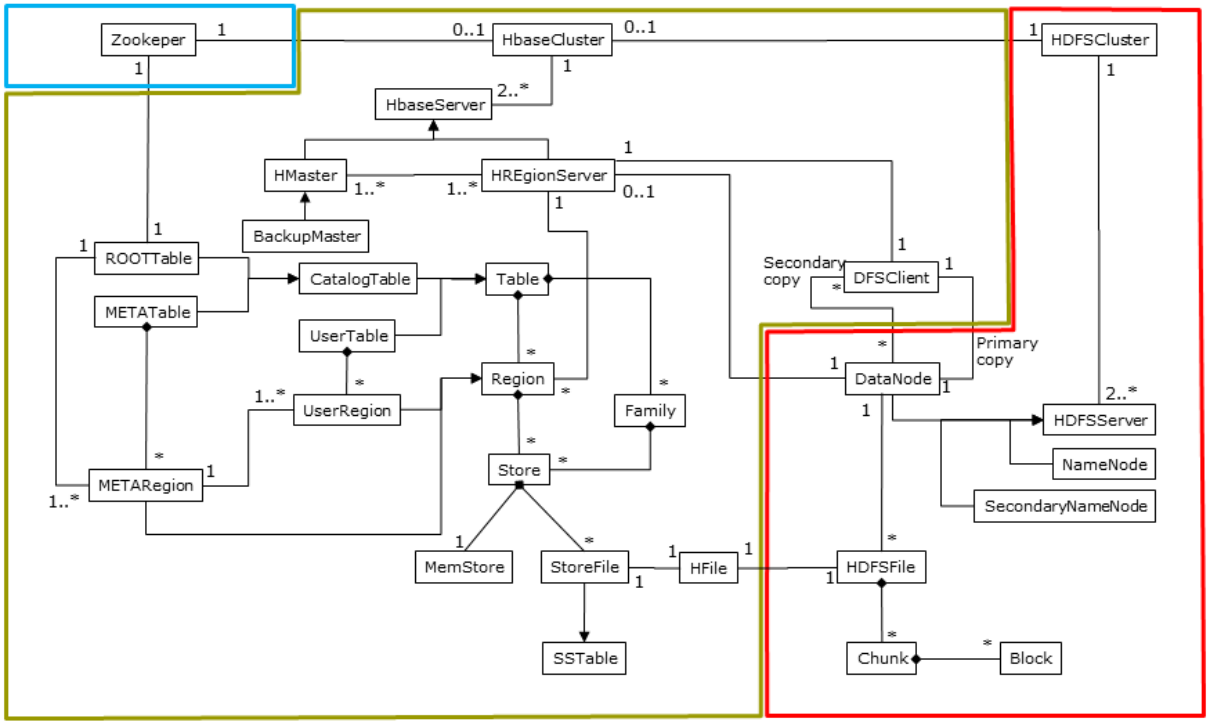


Figure 5.4: HBase and HDFS functional components and relationships

Figure 5.4 presents a UML diagram depicting how HDFS and HBase are coupled together. Besides how to achieve data locality, this also shows the core structure of HBase, which is important to remark before explaining the data locality. As shown in this figure, HBase tables are horizontally partitioned in regions that, in turn, are vertically partitioned (according to families) in stores. There is exactly one store per region and family. Data are physically stored in stores. First, in in-memory buffers (memstores), which are then flushed to disk as storefiles (a.k.a. SStables). Storefiles are represented as HFiles (having specific metadata), which are divided into HBase blocks. Finally, these storefiles need to be written in HDFS, so they are splitted into HDFS chucks (containing several HBase blocks each) and replicated across different DataNodes. Note that this is a logical schema and thus, the physical settings in terms of which HDFS blocks are stored are not depicted here (indeed, they depend on the cluster configuration).

In order to guarantee the data locality principle (i.e., a DataNode stores the HDFS blocks of the storefiles it holds as RegionServer), the control flow between HBase and HDFS is as follows. When a RegionServer writes on disk it asks to its DFS client to open a writer stream. As the RegionServer writes, the DFS client packages these data until it reaches the maximum HDFS block size. At this point, the DFS client communicates to the NameNode the need to materialize such block and it is the latter who decides where to place the master copy of such block (as well as its replicas). The NameNode applies an internal policy to do so that firstly checks if there is a DataNode running on the same node as the DFS client who asked for writing the block. If so, the *local* DataNode stores the master copy.

Relevantly, HBase also implements a cache to store recently read blocks. This way, HBase may save reading a

[2] `https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands`

block from disk if recently read and still cached. Last, note that HBase tuples can only be accessed using the HBase scan object, which retrieves tuples by means of the distributed tree index and thus, efficiently supports retrieving a single key or a range of (consecutive) keys (i.e., typical random accesses).

Finally, Zookeper[3] is "*a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.*" In HBase it is basically used to keep track of the distributed index. The Zookeeper points at the tree root table (*-ROOT-*) and whenever a tuple must be retrieved from HBase it finds out where to look for the tuple by exploring the tree. The HBase tree has three levels and it is stored as three regular HBase tables (one per level). At the first level there is the tree root. The next level corresponds to the regions of the catalog table (*.META.*), which points to RegionServers. Finally, the third level contains the regions where these data logically belong to.

For all this to work effciently, HBase implements three internal processes:

Flush happens when the memstore is full. It simply generates a new SStable structure in the disk persisting the current content of the memstore. Notice that if any of the keys present in memory at this point was previously persisted, it will result in two different versions.

Minor compactation happens periodically and runs in the background. It simply merges different SSTables resulting from independent flush executions of the same store. It does not necessarily merge all existing SSTables, so some obsolete versions may still survive.

Major compactation happens on demand (must be manually triggered). This merges absolutely all the SSTables of the same store, generating a single one, so guaranteeing that no obsolete version survives the process.

5.3 Data Management

We are going to analyse now how HBase provides solutions to each of the distribution challenges.

5.3.1 Data Design

Firstly, we need to see the mechanisms to distribute the data.

5.3.1.1 Fragmentation

In this section, we discuss how both horizontal and vertical partitioning are achieved in HBase in order to finally identify what factors are playing a key role in this matter so they are taken into account when tuning the system.

As discussed, HBase horizontal fragmentation distributes data across regions. When reading from HBase, the processing engine (e.g., MapReduce or Spark) splits the input data addressing each region to a different worker and thus, the number of regions (i.e., horizontal fragmentation) directly affects the degree of parallelism that can be achieved. The more the regions, the more parallelism we can reach. It is important to clarify that we place the discussion at the data design, hence we do not consider factors such as the number of mappers or reducers, since they do not belong to the database design stage, but to query execution time. Therefore, the factors we must take into account at this point are the the number of nodes (RegionServers), the number of families and the compression (we could also consider HDFS number of replicas).

HBase allows to manually fragment the tables instead of using an automatic policy. This resembles the situation for distributed RDBMS where data distribution is done at design time. However, the Hadoop ecosystem is thought to provide highly scalable settings and thus a static/predefined partitioning would not always be the best choice. For this reason, HBase can be configured to use different policies for dynamic/automatic partitioning and even provides tools to let us implement our own. For the sake of simplicity, we will focus on the default policy. This systematically checks if there is a storefile larger than a given threshold. If so, a new region split is triggered and a new fragment (i.e., region) is created. Importantly, if a storefile is split, all storefiles (i.e., family files) belonging to the same region will also split (even if they did not reach the set threshold) to preserve data locality.

$$split.threshold = min(R^2 mem.size, max.size) \tag{5.1}$$

[3] `https://zookeeper.apache.org`

Equation 5.1 shows how this threshold size is set. The splitting threshold is defined as the minimum of (i) a function of the number of regions in the corresponding RegionServer (R) and the maximum size of the memstore ($mem.size$), and (ii) a constant value $max.size$. The rationale behind such formula is to use $max.size$ as splitting factor in the long term. However, purely using a constant may lead to low performance in many cases. On the one hand, a large value would generate few fragments, and therefore very large amounts of data would be needed to exploit the parallelism of the Cloud. On the other hand, a small value would lead to too many fragments that would impact on the final execution cost due to the startup time of too many parallel tasks. Since setting the right $max.size$ would not be easy, this formula is thought to deal with this trade-off. Thus, at the beginning, the first element is used and data split at a faster pace (regardless of $max.size$). Eventually, that value will increase until surpassing $max.size$ after a certain amount of splits have taken place. Only from then on, the splitting step will remain constant.

We can consider that the memstore size $mem.size$ is fixed (default value is 128 MB), because disk storage will eventually be several times bigger than data in main memory. Accordingly, the partitioning strategy depends on a combination of the following factors: (i) the number of RegionServers, and (ii) the vertical partitioning and compression strategies that will impact on the growth pace of the storefiles. Let's assume a situation with five regions (i.e., fragments) in total:

- If the number of RegionServers is five and the regions are evenly distributed (i.e., every RegionServer stores one region), then $R = 1$ and according to the previous formula, the next region split will occur when any of the storefiles reaches:

$$split.threshold = 1^2 mem.size$$

- If the number of RegionServers is one, and therefore all the regions are stored in the same RegionServer, then:

$$split.threshold = 5^2 mem.size$$

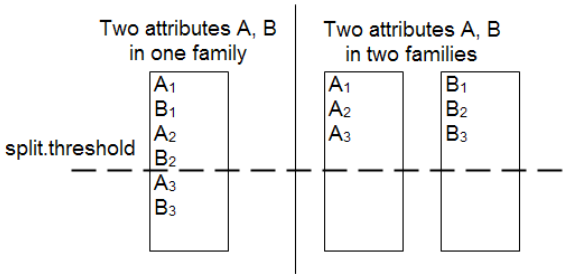


Figure 5.5: Effect of vertical fragmentation on region splits

Thus, the more RegionServers we have, the more regions are created. On the other hand, since each storefile contains exactly one family, the number of vertical fragments (i.e., families) determines the number of storefiles. Consequently, the larger the number of families the harder for a storefile to reach the splitting threshold, since, with less partitions, each family will contain more attributes and therefore it is faster for any of them to reach the splitting threshold (Figure 5.5 shows this graphically). Note that this implies that horizontal fragmentation pace depends on the vertical fragmentation design.

It is important to notice that compression has a similar effect on the storefile size. A strong compression makes the storefiles to use less space, so it takes more data to reach the splitting threshold. This effect is the other way round with lower compression (or no compression at all). Summing up, since $max.size$ and $mem.size$ are two constant values, the number of RegionServers set the split threshold, whereas the vertical fragmentation and the compression algorithm used (if any) determine how fast the split threshold is reached (e.g., with no compression and one family the split threshold will be reached faster than with ten families and a heavy compression algorithm).

5.3.1.2 Allocation

An important issue the reader may note about this fragmentation policy is that it does not guarantee an even distribution of data, which completely depends on the choice of key with regard to insertion order. More precisely,

such even data distribution can only be assumed to take place eventually, when the constant value $max.size$ is used as main splitting factor (bear in mind that distribution in HBase is performed based on the number of regions each RegionServer holds). To mitigate this effect, HBase provides a balancer process that runs periodically (frequency can be configured in `hbase.balancer.period`). This moves regions around to balance workload in the cluster, and guarantee a similar number of regions in each regionserver. Indeed, HBase does not take into account the amount of data each RegionServer contains when distributing, but the number of regions. Furthermore, the first argument of Equation 5.1, which is the main splitting factor in the short time, is quadratic and may lead to sensible differences in the data distribution between nodes for small differences in the number of regions. The poor performance of HBase and MapReduce when distributing data has already been highlighted (e.g., see [DN14]).

5.3.1.3 Replication

On the other hand, regarding replication, it completely relies on that of HDFS mechanism for availability (i.e., HFile replicas can be enabled like in any other HDFS file, and would be used in case of failure of the data node hosting the primary copy). Nevertheless, notice that this does not affect memstores and consequently such replicas would be ignored in generic query processing. Writes will always go to the primary copy, but to truly have replication being used in the queries, we should explicitly enable HBase read replicas.[4] With this, region servers periodically replicate their data, which can then be used to serve queries regularly. The consistency level of a query can be set, so that secondary regions (potentialy stale) are ignored.

5.3.2 Catalog Management

As already explained in Section 5.2, the internal levels of the tree structure (i.e., somehow the catalog) are also stored as tables and follow the same structure in regions as data. Consequently, everything explained about data design in the previous section applies to this as well. However, we are going to briefly discuss two particularities of HBase catalog.

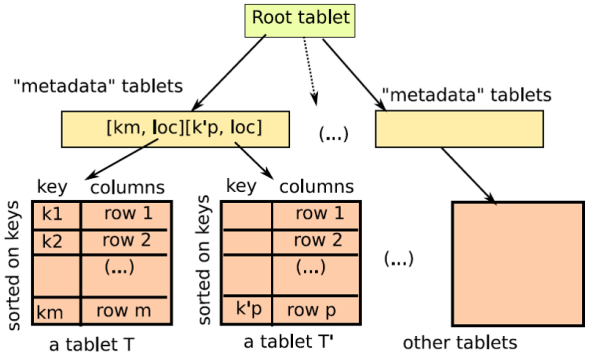


Figure 5.6: Tree structure of HBase with three levels, from [AMR+11]

5.3.2.1 Internal tree nodes

In general, there is not any limitation in the number of levels in the tree structure, but is it assumed to have three. Thus, Figure 5.6 depicts them. The top level is know as Root and the one below Meta. The former points to the latter, and the latter points to the user data. Actually, the three of them (i.e., Root, Meta and user data) are independent tables, the only difference being that the first two store locations instead of user data.

Just making few numbers, it is easy to see why it is not necessary to have more than three levels. Let's assume that the Root table only occupies one memstore of $128MB$ (i.e., it never required to be flushed to generate other SSTables). Then, if each [key, location] pair occupies $128bytes$, we can point from there to 10^6 regions at the Meta level. Now, under the same assumption of only using the $128MB$ of a single memstore each, we could point to 10^{12} user regions from there. Then, only using memstores also at the user level, we could store $128 \cdot 10^{12} MB = 128 \cdot 10^3 PB$, which looks like more that enough for any single table we can imagine today.

[4] `https://docs.cloudera.com/runtime/7.2.6/hbase-high-availability/topics/hbase-read-replicas.html`

5.3.2.2 Metadata synchronization

HBase implements a client cache similar to that of HDFS (see Section 4.3.2) to avoid constantly disturbing the coordinator (also the intermediate regions). Thus, only the first time a key is requested by an application, the request needs to go down the tree structure of metadata. Eventually, some regionserver will send the corresponding data to the client and this will take note in the cache of who did this. In successive requests, that key will be found in the client cache and the request will be directly addressed to the right regionserver.

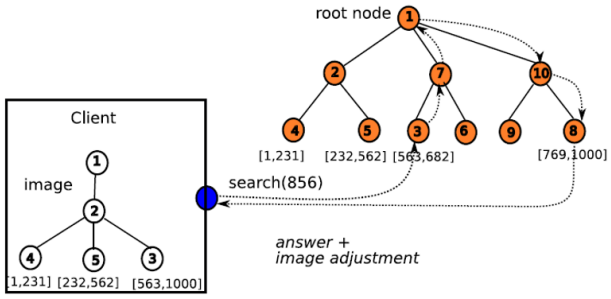


Figure 5.7: HBase client cache mistake compensation, from [AMR+11]

Nevertheless, the tree structure of HBase is more volatile than that of HDFS directory (i.e., splits and merges of regions happen more frequently than rewrites of files) and also being multilevel (i.e., Root and Meta) complicates its management. Thus, it can happen that when the application finds the key in the client cache and requests it to the regionserver registered there, such region server does not have that key anymore. If this is the case, as exemplified in Figure 5.7, it will be the region server itself that will scale the request up the tree structure to its parent. In the worse case, the key would not be under the parent either and this will propagate the request to the Rootregion. Since the Rootregion has the information of the whole domain of keys, it is guaranteed that it will be able to forward the request down the tree to the appropriate Metaregion, and this to the user regionserver that now has the key, which will directly send the corresponding value to the client. Since we are assuming that the tree has three levels, it means that this compensation actions will require at most four extra calls between the different regionservers (i.e., two upwards and two downwards).

5.3.3 Transaction Management

As it is typical in NOSQL systems, HBase does not completely support ACID transactions, but only a very limited interpretation.[5]

Atomicity: It is only guaranteed at the row level. The classical concept of transaction affecting different rows or even different tables in RDBMS does not exist in HBase. Thus, it only guarantees that if data us put in different families for the same row. This will happen atomically (either all families or none).

Consistency: Actually, it does not provide any kind of integrity constraints (neither checks nor foreign keys). As previously said, replication management at the disk leval completely relies on that provided by HDFS underneath. Thus, the configuration of persistent data is eager/primary-copy. However, the synchronization of memstores is managed by HBasee and would follow a Lazy/Primary-copy model.

Isolation: It offers the read committed ISO isolation level by locking all families at once for the same row (notice that all families for a given region are always in the same machine). However, as already said, there is not any way to wrap a set of multi-row operations into a single unit. Consequently, there is not any guarantee of snapshot isolation (i.e., the scan of a table corresponds exactly to its state at a given point in time), since during the execution of the scan rows can appear or disappear (e.g., in different machines), so that its result does neither correspond to the state before nor after the operation.

Durability: Before confirming any modification in a table, the operation is annotated in the log file (i.e., it follows the Write Ahead Log protocol), which guarantees it is never lost.

[5] `https://hbase.apache.org/acid-semantics.html`

5.3.4 Query Processing

The global execution manager routes the requests through the three levels of metadata, which clearly provides inter-query parallelism, because different queries can go to different regions (hence servers) in the Cloud. This can be amplified if we enable read replicas (as explained in previous section). Nevertheless, intra-query parallelism is very limited since HBase does not provide parallel implementation of the scan operation. Thus, increasing the number of replicas in HDFS will increase the availability and reliability, but not parallelism. If we really want to perform a parallel scan, the domain of the key must be divided in different ranges at the application level and the corresponding requests launched from different threads (this actually resulting in inter-query parallelism). Then, each request will eventually reach a local execution manager, which will sequentially go through all the required stores depending on the existing families in the table. Firstly, it would check the memfile taking any version of the value found there. Then, independently of finding anything in memory or not, store files need to be checked, because they might contain other versions. However, some accesses to the storefiles could be saved thanks to (1) the existing bloom filters, or (2) if ranges of keys are disjoint after a compactation.

Declarative query support can be provided through HiveQL[6], a query language resembling SQL for unstructured data such as those stored in HBase implemented by the independent project Apache Hive[7]. When defining Hive tables over HBase we specify the column families and qualifiers where each attribute is stored. Besides basic data selection functionalities, HiveQL offers new operators to deal with complex data types (e.g., arrays or nested data), such as `LATERAL VIEW` or `EXPLODE()`.

[6] `https://cwiki.apache.org/confluence/display/Hive/LanguageManual`
[7] `https://hive.apache.org`

Chapter 6

Document Stores

The Relational model was defined as an abstraction level to gain independence of the file system and any internal storage structure (see [Cod70]). Thus, we could gain flexibility and interoperability without losing efficiency by following a fixed tabular representation and some normal forms (see [AHV95]). Indeed, the first normal form (1NF) established that attribute domains had to be atomic (i.e., they could be neither compound-complex structures nor arrays). However, a rigid tabular structure is not adequate in modern agile software development, where the schema is under continuous evolution. Moreover, the Web is a powerful medium for human communication and an extraordinary source of information. Consequently, it has become a popular knowledge base, where people add documents (private, educational and organizational) and navigate through their variable and diverse contents. For scalability reasons, one important challenge is to distill those documents and extract valuable knowledge from them.

There exist multiple formats for information sources, ranging from unstructured data to highly structured. As explained in [Abi97], the term semi-structured data emerged to describe data that has some structure but neither regular, nor known a priori to the system. Hence, semi-structured documents are self-describing. Conversely, a structured database distinguishes schema and instances, fixing the former, which is a set of attributes, each with a concrete domain, while the later is a tuple of values that belong to the corresponding domain in the previously declared schema.

From a mere technical perspective, a well-known problem of RDBMS is the impedance mismatch, defined as the overhead generated by transformations from internal structures to tables, and then into programming structures (see [Amb03]). The development of NOSQL systems, which adopt more flexible data representations, allowed to overcome such impedance mismatch too (see [SF12]).

Indeed, some of such semi-structured documents (e.g., JSON), are directly mapped from disk to memory, and thus break the data independence barrier. This is additionally achieved by breaking 1NF, allowing typical programming nested structures and arrays in the attribute values (e.g., MongoDB encourages denormalization[1]). Furthermore, such semi-structured formats, also allow to skip schema declaration, which is beneficial in agile software development and quickly evolving programming environments (see [SS20]). Nevertheless, it is not clear whether denormalization and schemaless is a conscious design choice, or merely a paradigm imposed by the limitations of NOSQL systems. Yet, the flexibility offered by NOSQL comes at a price, where each one of the associated design choices may widely change their physical representation, and thus profoundly impact performance. Practitioners have typically ignored this, and today make binary design decisions based on rules, and programming needs with no overall view of the system needs (see [Moh13]). Thus, it is vital to consider the benefits and drawbacks posed by these different alternatives during the design process (see [BL11]). As already discussed in Chapter 2, Relational and semi-structured database models, are not a simple binary choice, but a continuum of options with different degrees of (de)normalization. This idea is pursued by the co-relational model in [MB11], which entails a wide range of complementary database design possibilities.

[1] https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-2

6.1 Semi-structured database model

The term semi-structured describes data that have some structure but is neither regular nor known a priori (see [Abi97]). The most dominant semi-structure model implementations are eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) (see [TADP21] for a comparison of both).

eXtensible Markup Language (XML). documents have been adopted as standard for data interchange, enabling the integration of heterogeneous information sources. A *well-formed* XML document is a document that conforms to the XML syntax rules in [W3C04] (roughly, markups nest properly and attributes are unique). Moreover, a *valid* XML document is a document that is *well-formed* and also conforms to the rules of its grammar. A Document Type Definition (DTD) contains the declarations that provide such grammar for a class of documents. It determines the *element*s and *attribute*s that appear in a document (i.e., the name, type and constraints on every *element* and *attribute*). This is really important for the interchange of documents, since it represents the meaning of data. However, some automatically extracted documents (maybe coming from HTML) lack even this simple kind of schema, which is absolutely optional. As defined in [W3C04], an XML document primarily consists of a nested hierarchy of *elements* with a single root. *Elements* can contain character data and *child element*s. In both cases, the *element*s can have *attribute*s. The structure of *child elements* consists of a *sequence* list of *element*s. The standard states that *element*s in a *sequence* must be ordered, but this is rarely considered to be relevant in existing implementations. The *choice* construct in a DTD indicates that one, and only one, *element* in the *choice* list of contents should appear in the document (i.e., alternative elements).

JavaScript Object Notation (JSON) format is completely equivalent to XML, with only some change in the terminology. Thus, a JSON document consists of a nested hierarchy of key-value pairs with a single root. Child documents are an unordered sequence list of pairs with optional presence. Hence, as for XML, JSON documents are self-descriptive, and do not require a schema declaration (i.e., this is optional), despite a known structure facilitates storage and encourages queries.

6.1.1 Data structure alternatives

Considering those definitions, we present representational differences between semi-structured and structured data (i.e., equivalent alternatives to represent some datum exploiting the characteristics offered by each of both models), and discuss their potential impact on storage size, data insertion, and query performance (see [HNA20] for more details). More specifically in the next sections, we discuss the following issues:

Schema variability. A common fixed schema is defined for all instances in structured data, but potentially different schema is specific to every document in semi-structured. This entails:

1. Metadata embedding
2. Attribute optionality

Schema declaration. *A priori* rigid schema declaration is mandatory in structured data, but is just optional and more flexible (grammar-based) in semi-structured. This includes the declaration of:

1. Structure and data types
2. Integrity constraints

Structure complexity. Only flatten instances containing atomic values are allowed in structured data, but complex nesting can be used in semi-structured. This includes the representation of:

1. Nested structures
2. Multi-valued attributes

6.1.1.1 Schema variability

A common schema is defined for all instances in structured databases, but in JSON, there may exist potentially different document schemas inside the same collection of documents. Here, we focus on comparing alternative ways to represent such schema.

JSON

```
{ _id: 123,
 A1: "x",
 ...
 An: "x"
}
```

Tuple

_id	A_1	...	A_n
123	"x"	...	"x"

Figure 6.1: Alternative representations of Metadata

6.1.1.1.1 Metadata embedding Representing different schemas across JSON documents entails embedding their metadata into each instance (see Figure 6.1). This clearly impacts negatively the size of the database and consequently query performance. The more attributes are present, the more metadata (i.e., attribute names) will be embedded into each document. Additionally, the ratio between the size of data and metadata is clearly an important factor to consider (i.e., attribute name length w.r.t. its values). Thus, we need to consider (a) the absolute amount of metadata by analysing the number of attributes, and (b) the relative amount of metadata by considering the ratio between value length and attribute name length).

JSON-Absent

```
{ _id: 123
}
```

JSON-NULL

```
{ _id: 123,
 A1: null,
 ...
 An: null
}
```

JSON-666

```
{ _id: 123,
 A1: 666,
 ...
 An: 666
}
```

Tuple-NULL

_id	A_1	...	A_n
123	null	...	null

Tuple-666

_id	A_1	...	A_n
123	666	...	666

Figure 6.2: Alternative representations for optional attributes

6.1.1.1.2 Attribute optionality Another feature of the semi-structured data model is the possibility to skip the representation of an attribute in the absence of its value (as the case of *JSON-Absent* in Figure 6.2). However, it also supports to, either use a special value outside the domain (as in *JSON-NULL*) or use a specific value inside the attribute domain (as in *JSON-666*). Notice that in a Relational representation, as the schema is fixed and common to all instances, only the last two options are possible (as in *Tuple-NULL* and *Tuple-666*, respectively). The impact on space and performance of these representations will vary depending on the percentage of absent/present values for the attribute.

6.1.1.2 Schema declaration

In order to benefit from schema declaration and validation in semi-structured databases, one must adopt additional constructs. *JSONSchema* is a JSON-based schema language that allows to constrain the shape, types and values of JSON documents (see [PRS+16]). Here, we should consider the impact of both structure plus data type declaration, and integrity constraint (IC) validation separately.

JSON Type

```
{ _id: 123,
 A1: k,
 ...
 A64: k
}
```

```
{ "type": "object",
 "properties": {
  "A1": {
   "type": "number"
  },
  ...
  "An": {
   "type": "number"
   },
  required: ["A1",...,"An"]
  }
}
```

Tuple Type

_id	A_1	...	A_{64}
123	k	...	k

Figure 6.3: Alternative representations of structure and data type validation

6.1.1.2.1 Structure and data types To validate structure and data types, *JSONSchema*[2] uses the `properties` key. For each attribute, it is possible to specify its data type, which can be either a primitive or complex object. Furthermore, the `required` key represents an array enumerating the list of expected attributes. Figure 6.3 depicts the exemplary document patterns considered. Clearly, this declaration has no impact on database size, since it does not grow with instances. However, it has a cost on insertion, corresponding to validating presence and domain, and on the other hand, it could potentially benefit query time by saving an explicit casting and type conversion.

JSON-IC

```
{ _id:    { "type": "object",
   123,     "properties": {
 A1: k,       "A1": {
 ...            "type": "number",
 A64: k         "minimum":−k′,
}               "maximum: k′},
              ...
              "An": {
                "type": "number",
                "minimum":−k′,
                "maximum: k′}
            }
          }
```

Tuple-IC

_id	A_1	...	A_{64}
123	k	...	k

```
ALTER TABLE T ADD CONSTRAINT
val_A1 CHECK
(A1 BETWEEN −k′ AND k′);
...
ALTER TABLE T ADD CONSTRAINT
val_An CHECK
(An BETWEEN −k′ AND k′);
```

Figure 6.4: Alternative representations of Integrity Constraints (IC) validation

6.1.1.2.2 Integrity constraints Besides the data type validation mechanisms, `JSONSchema` also offers means to represent integrity constraints for attributes. Here, as depicted in Figure 6.4, we focus on enforcing ranges of values. In Relational databases, this is achieved via `CHECK` constraints. As above, this has no impact on the size of the database but will have some on the insertion since it has to be checked before accepting the data. Despite this, it might also be used to perform some semantic optimization at query time, which we can consider as technology-specific (i.e., not directly dependent on the data representation).

6.1.1.3 Structure complexity

An RDBMS conforms to 1NF, yet a semi-structured one relaxes such restriction, which allows storing nested and multi-valued data. Here, we discuss the impact of different complexity degrees on data according to that.

JSON-Nest

```
{ _id: 123,
 L1: {
  ...
  Ln: {
   An+1: k }
   ...
   }
}
```

JSON-Array

```
{ _id: 123,
 A: [1,...,n]
}
```

JSON-Attrib.

```
{  _id: 123,
 A1: k,
 ...
 An: k
}
```

Tuple-Array

_id	A
123	[1,...,n]

Tuple-Attrib.

_id	A_1	...	A_n
123	k	...	k

Figure 6.5: Representations of nesting structures and multi-valued attributes

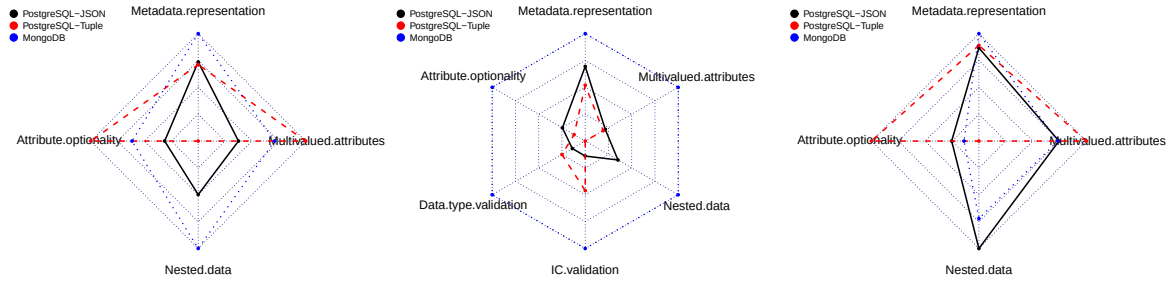
6.1.1.3.1 Nested structures Documents allow to explicit into a data structure conceptually independent objects, which are accessed using dot notation. Yet, it is unclear what is the impact regarding size (i.e., with an increasing number of brackets in the document), and on querying such structures. To explore this, we should consider a range of levels and attributes (*JSON-Nest* in Figure 6.5). Thus, for every level we add we need to consider the required extra characters (i.e., ":", "{", and "}"). Such nesting is simply not possible in the Relational model.

[2] `https://json-schema.org`

6.1.1.3.2 Multi-valued attributes Variable-length multidimensional arrays also violate 1NF's restrictions. Nevertheless, modern object-relational DBMSs have adopted variable-length multidimensional arrays as data type, an aspect present in semi-structured JSON documents by definition. Yet, it is unclear what is the impact of managing such types. On bounded arrays, one could argue that it might be better to store each position as an independent attribute, as depicted in Figure 6.5, where we distinguish, for both JSON and tuples, *array* and *multi-attribute* alternatives. Multi-valued attributes could also be stored in a separate normalized table, however such independent structure would compete for memory resources, heavily impacting insertion and query. As before, we consider that eviction policies affecting that memory competition are technology-specific and not purely dependent on the data design (see [HAVZ20] for a performance analysis of caching in MongoDB).

6.1.2 Impact analysis of alternatives

For each representational difference, we present now a discussion of the consequences in performance. Figure 6.6 summarizes all results with regard to storage space, load time, and query time. For this, we calculated the average of all measurements per representational difference for each of the three options (i.e., Tuples and JSON in PostgreSQL, and storing JSON in MondoDB). Since data follows different patterns in each case, we separately min-normalize per case (e.g., $min(PG-JSON; PG-Tuple; M-JSON)/PG-Tuple$) and plot them all in the corresponding radar chart. This means values further away from the center of the radar are better than the ones closer, and the bigger the area of the polygon, the better the system performs.



(a) Storage (b) Insertion time (c) Query time

Figure 6.6: Multidimensional view of experimental results

According to Figure 6.6a, storing tuples takes the least amount of space in all cases except metadata representation. On interpreting this, we acknowledge the impact of the ratio between metadata and data, which is fixed to be relatively high in all experiments. Thus, attribute names should always be encoded in JSON to shorten them as much as possible and improve that ratio. Obviously, this is more relevant, for example, if values are numeric than if they are strings (the former requiring less space, in general). Within JSON, PostgreSQL storage size is much larger than MongoDB in all the cases, due to the different encoding of integers (64-bits vs. 32-bits).

According to Figure 6.6b, it is clear looking at PostgreSQL that loading JSON is faster than tuples, except for data type and integrity constraint validations. However, it is important to note that the validation of JSON was carried out through a third-party plugin, which definitely impacts the results. MongoDB being a native document store, has a clear advantage over PostgreSQL JSON storage in loading data (at the end of the day, JSON is stored as a column in a PostgreSQL table), beating even tuple storage in the validation dimensions. This, however, can come not only from using JSON format but from other DBMS characteristics (e.g., lack of fully ACID transactional support in MongoDB).

Finally, Figure 6.6c depicts that tuples, in general, perform better in queries. This is so because they use less space, in general, and benefit from validation at insertion time. Thus, we can see that when the space-saving is lost depending on the data-metadata ratio, so the benefit is mostly lost at query time, as well. Nonetheless, JSON representation is at a disadvantage, as each of the documents needs to be parsed and processed on demand. Consequently, we should consider the trade-off between the pressure of fast ingestion and the long term benefit of recurring queries. It is also interesting to see that even though the storage size of JSON is larger in PostgreSQL, this is still faster than MongoDB. We believe this fact results from the differences in how query engines handle the calculations. PostgreSQL benefits here from the well-optimized aggregation operations in the Relational engine, which data stored in JSON format also have access to.

6.2 MongoDB Architecture

The access atom in MongoDB is the document, which always has associated an identifier. This identifier can be system-generated, or provided by the user. Thus, we can see a document store like a Key-Value store, where the value has some structure. This is important, because such structure, even if flexible, still allows to define secondary indexes, which is not possible in Key-Value stores.

A set of documents conform a collection. Unlike in RDBMS tables, documents inside the same collection can be heterogeneous. At most, we can associate a grammar to the documents in the form of JSONSchema. Data types can be included in the grammar, and we can define attributes that reference identifiers of documents in other collections. Nevertheless, referential integrity (a.k.a., foreign key) is not going to be enforced.

The available set of commands in MongoDB shell[3] is much richer than that of HBase, simply because it provides more query mechanisms and alternatives. However, it is still far away from all the power of standard SQL.

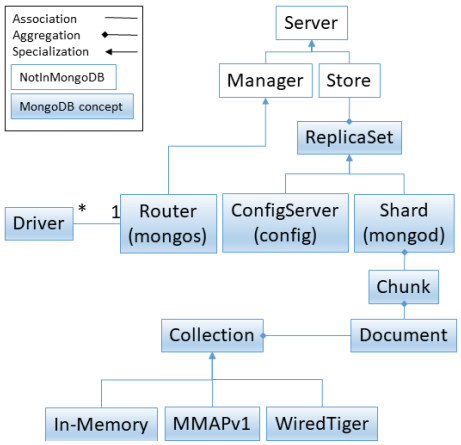


Figure 6.7: MongoDB functional components

Regarding the functional architecture (summarized in Figure 6.7), we must first distinguish between machines that contain data, organized in replica sets (of typically three machines), and those that purely route queries, known as *mongos*. Among the former, we can distinguish the *Config Servers* that contain the global catalog, which keeps track of existing *Shards*, who actually store data. There is also a *Balancer* process inside the primary config server in charge of detecting unbalanced shards and moving chuncks from one shard to another. This allows shards to split or migrate chunks of data between different machines.

On the other hand, *mongos* split the queries and merge back the results. To make it more efficient and avoid disturbing the coordinator, as usual, they maintain a cache of shards, which is lazily synchronized. Typically (but not necessarily), *mongos* sit in the client machine to avoid network traffic.

Internally, JSON documents are serialized, which includes encoding numerical values, for example (compression can also be enabled). This internal format is called BSON (a.k.a. JSONB in PostgreSQL, which obviously slightly differ in the implementation). Several documents are grouped in a chunk, which is the distribution unit (i.e., equivalent to the homonym in HDFS). There are three engines that can manage these chunks, namely In-memory (does not guarantee persistence), MMAPv1 (implements a consistent hash) and WiredTiger (implementing a B-tree).

WiredTiger is an independent storage engine integrated in MongoDBv3 (it became the default storage in v3.2). It allows MongoDB to enjoy block compression, as well as key prefix and value dictionary. However, even if WiredTiger provides both horizontal and vertical fragmentation, only the former is compatible with MongoDB. Morover, it also has an implementation of an LSM-tree, but it is currently not supported by MongoDB,[4] either.

[3] `http://docs.mongodb.org/manual/reference/mongo-shell`
[4] `https://jira.mongodb.org/browse/SERVER-18396`

6.3 Data Management

Besides a semi-structured model, the main feature of MongoDB is automatic and transparent scalability. It is conceived to live in the Cloud and elastically use the required resources without much human intervention.

6.3.1 Data Design

In the next sections, we are going to see the possibilities that MongoDB offers to distribute data.

6.3.1.1 Fragmentation

Horizontal fragmentation (called sharding in this context) is one of MongoDB main features. Both range-based and hash-based are possible, depending on the chosen engine being respectively WiredTiger or MMAPv1. However, despite being implemented in WiredTiger, vertical fragmentation is not available in MongoDB.

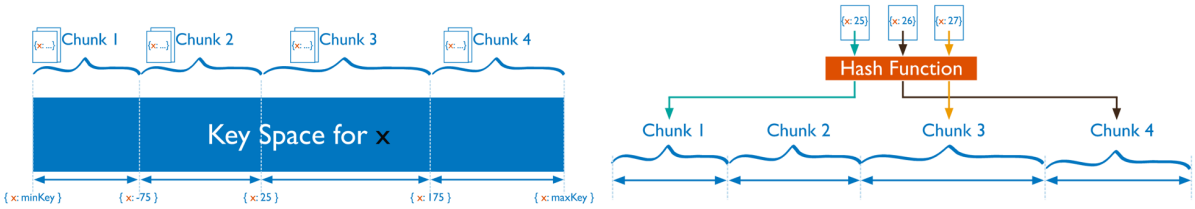


Figure 6.8: MongoDB kinds of horizontal fragmentation

Figure 6.8 sketches both options. At LHS, we can see the range-based one, where the chunk of every key is decided based on some ranges of values. Oppositely, at RHS, we see that there is a hash function which returns the chunck for every key. In general, the key is a mandatory attribute in all the documents of the collection, and must be indexed. It can be chosen by calling `sh.shardCollection(<namespace>, <key>)`.

6.3.1.2 Allocation

With either one option or the other (i.e., range or hash-based fragmentation), the chunk where each document sits is dynamically decided. Thus, a threshold is defined, so that when a chuck grows beyond it, a new chuck is created. In case this happens, the following process is triggered:

1. A new chunk is created in an underused shard
2. Per document requests are sent to the origin shard
3. Origin keeps working as usual (changes made during the migration are applied a posteriori in the destination shard)
4. Changes in the catalog are made in the config servers (this enables the new chunk)
5. Chunk at origin is dropped
6. Query routers are eventually synchronized with the new information in the config server

6.3.1.3 Replication

Replication in MongoDB is based on *Replica sets*. These are sets of mongod instances (typically three) that act coordinately. Thus, a shard siting in a replica set means that its data is mirrored in all the nodes that belong to that replica set. Since replica sets are disjoint, this means that scaling by sharding results very expensive in terms of the number of machines (since you multiply shards by the number of replicas). It may be better to simply add more memory to a single machine.[5]

[5] `https://developer.mongodb.com/article/everything-you-know-is-wrong`

6.3.2 Catalog Management

As expected, the catalog information in MongoDB (mainly composed by the list of chunks in every shard) is treated like any other piece of data. Indeed, the data is stored in a replica set, and consequently enjoys alls its synchronization benefits and consequences (see Section 6.3.3). The only specificity is that its information is cached in the routers (similarly to HDFS directory and HBase internal nodes information). The behaviour in MongoDB is also Lazy/Primary-copy.[6]

6.3.3 Transaction Management

As already mentioned, all data assigned to a replica set is mirrored in all its nodes. One of them acts as the primary copy, and is the only one that accepts modifications of the data. The others only accept reads from users and periodically connect to the primary to receive the changes. The client application can specify which of the replicas wants to read. Thus, the primary copy should be explicitly specified to guarantee that the last version of data is always retrieved. Alternatively, we can fine tune the readconcern[7] to specify how many copies need to be read before returning data to the client application. Conversely, we can also fine tune the writeconcern[8] to specify how many copies need to be written before confirming the writing to the client application.

Notice that implementing a mechanism allowing only writes in the primary copy makes this a single point of failure. To mitigate such problem, MongoDB implements a heartbeat mechanism. Thus, in case of failure of the primary copy (i.e., not producing a heartbeat for more than ten seconds) one of the secondary ones is automatically promoted so the system can keep on accepting modification in this one. The protocol is based on a consensus mechanism known as Paxos (see [Lam98]).

As a summary, we can say that MongoDB implements a very relaxed version of ACID properties:

Atomicity: The atom is a document, and transactions across different documents and even collections are allowed (since version 4.2).

Consistency: In general, MongoDB does not provide any schema, and hence, no integrity constraints can be enforced. However, you can optionally define a JSONSchema, which enables these features. Anyway, foreign keys are not available. Regarding replica synchronization, it implements a Lazy/Primary-copy mechanism.[9] Only the primary copy of a replica set can be modified, and eventually changes will be propagated to the other sibling servers.

Isolation: The granule of operations is the document, which means that different users cannot change the same document at once. However, they can modify different documents inside the same collection. For this, WiredTiger uses MultiVersion Concurrency Control, thus providing snapshot isolation. This means that in the presence of concurrent changes it will always provide a view of the collection which is consistent at a given point in time.

Durability: Writing requests are immediately confirmed to the client applications. Obviously, this speeds up writings, but comes with the risk of loosing some of these confirmed changes in the unlikely case of system failure between the confirmation and the writing is actually confirmed. By default, the time between a writing operation and the corresponding change being really registered in the disk is $100ms$.[10]

6.3.4 Query Processing

MongoDB offers three different query mechanims. Firstly, you can simply retrieve one (i.e., *findOne*) or many documents (i.e., *find*) given a document pattern,[11] using a JavaScript API. For more elaborated processing, it also offers the Aggregation Framework, which allows to define a pipeline of operators for documents to go through it.[12] For example, the first operator in the pipeline can filter documents with a certain characteristics, and the next

[6] https://github.com/mongodb/mongo/blob/master/src/mongo/db/s/README.md
[7] https://docs.mongodb.com/manual/reference/read-concern
[8] https://docs.mongodb.com/manual/reference/write-concern
[9] https://docs.mongodb.com/manual/core/replica-set-sync
[10] https://docs.mongodb.com/manual/core/journaling
[11] https://docs.mongodb.com/manual/tutorial/query-documents
[12] https://docs.mongodb.com/manual/core/aggregation-pipeline

element groups them based on another characteristic. Finally, the third available mechanism is a MapReduce stile function call.[13]

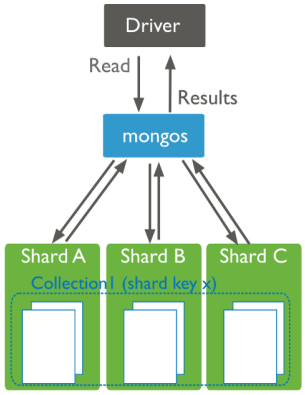


Figure 6.9: MongoDB routing mechanism, from MongoDB web site

As sketched in Figure 6.9, independently of the query mechanism being used, when data is sharded in a cluster of machines, the client application (a.k.a. Driver) has to contact the router (a.k.a. mongos), which plays the role of global query manager.[14] Hence, this is in charge of dividing the query among the different shards, and later merging the results. However, the mechanism is rather simple and just pushes the first selection and projection to the shards, to then move all the documents back to the router that executes the rest of the query or pipeline. Therefore, we can say that it obviously provides inter-query parallelism, since different routers and replicas can serve different users, but not inter-operator parallelism, since all are run in the router after the first operation in the shards is finished. Moreover, it offers also intra-operator parallelism in case of static fragmentation. Indeed, different shards would serve different pieces of the collection in parallel for the same query. It is important to remember at this point that MongoDB does not offer join operations, which would require more sophisticated parallelization mechanisms.

On the other hand, MongoDB offers different kinds of indexes (i.e., B+, hash, geospatial and textual), which can be multi-attribute, or be even defined over arrays. Nevertheless, the optimizer is not cost-based. Instead, as shown in Figure 6.10, it relies on a caching mechanisms that keeps track of how the latest queries were executed[15]. On getting a query, the cache is visited to see it there is a matching entry. If found, it is used to generate a plan for the current one, which will be then tested (i.e., executed) for some time. If the generated plan takes too long or there is not any query pattern in the cache matching the current query, the system launches different executions in parallel using alternative access paths (i.e., different indexes). Eventually, one of such executions will finish and the others will be killed. The executed one will be the one kept in the cache.

[13] https://docs.mongodb.com/manual/core/map-reduce
[14] https://docs.mongodb.com/manual/core/sharded-cluster-query-router
[15] https://www.docs4dev.com/docs/en/mongodb/v3.6/reference/core-query-plans.html

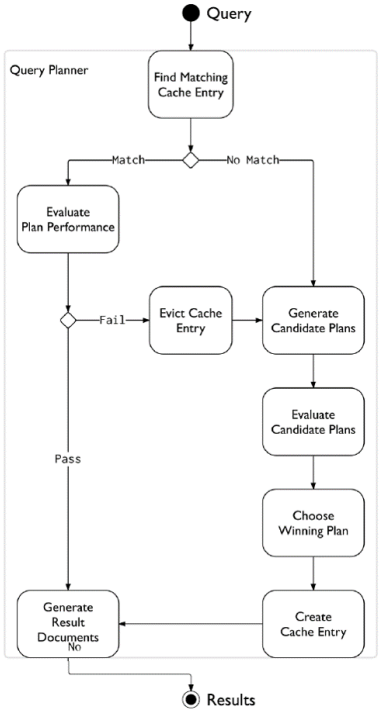


Figure 6.10: MongoDB query optimization, from MongoDB 3.6 manual

Chapter 7

New Relational architecture

The classical RDBMS architecture was based on the System-R project of IBM[1] in the late 70's. At that time memory was very scarce, slow and expensive. Thus, the main concern was how to efficiently manage it and swap irrelevant data to the much cheaper and steady hard disk.

In 40 years, hardware has evolved a lot. Nowadays, memory is much cheaper and communication much faster. Today, it is easy and relatively cheap (few tens of thousands of Euros) to have tens of interconnected machines with tens of GBs each, to amount one TB or even more memory altogether (30machines$\times$32GB=960GB$\simeq$1TB). Just to weight this up, we can consider the scalability factor of a realistic benchmark like TPC-C[2] based on the number of warehouses of a realistic company. Thus, each of these warehouses is considered to need 100MB, so that a company with a thousand of those would require a database of only 100GB. Hence, that database could be hosted purely in the memory of a cluster like the one described previously.

The first such prototype breaking with the traditional System-R architecture and allowing in-memory processing, was SanssouciDB in 2011, whose details are explained in [PZ11]. Such engines rely on vertical fragmentation and have shown excellent performance for read-intensive workloads. Nevertheless, this is not always the best modeling choice and adaptive systems have also appeared that dynamically exploit vertical or horizontal layouts depending on the workload. Consequently, DBMS vendors started revamping the original horizontal disk-based architecture and creating new releases that rely more on hybrid in-memory layouts like SAP HANA[3], Vertica[4], IBM BLU[5] or Oracle In Memory[6].

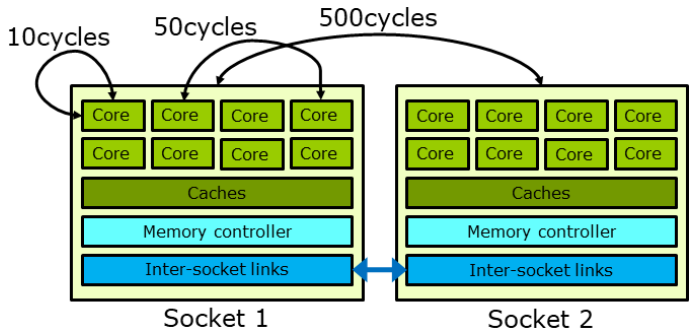
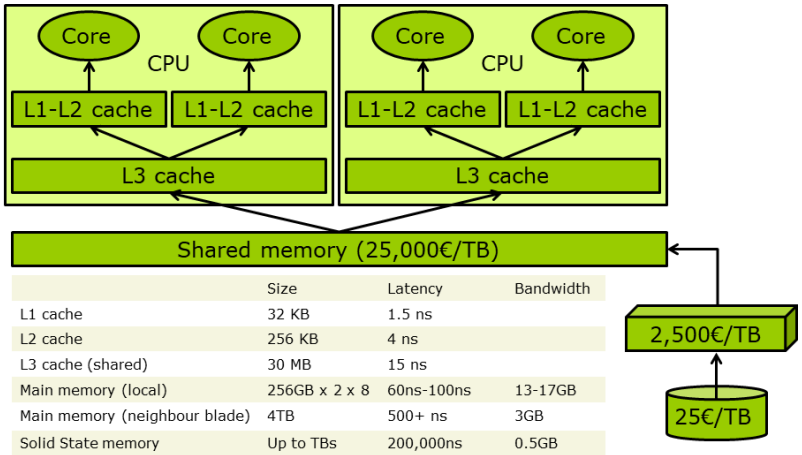


Figure 7.1: Non-Uniform Memory Architecture (NUMA)

[1] https://en.wikipedia.org/wiki/IBM_System_R
[2] http://www.tpc.org/tpcc
[3] https://www.sap.com/product/technology-platform/hana.html
[4] https://www.vertica.com
[5] https://en.wikipedia.org/wiki/IBM_BLU_Acceleration
[6] https://www.oracle.com/database/technologies/in-memory.html

7.1 In-Memory data management

Non-Uniform Memory Access (NUMA)[7] is a kind of hardware architecture characterized by having different access time depending on which processor accesses which memory address. This kind of hardware facilitates scaling up by growing a single machine (in front of the more popular scale-out approach, characterized by adding commodity machines to a computer cluster, like in Cloud computing). Thus, as depicted in Fig. 7.1, we can have different processors in different sockets. Each processor has different cores and different cache memories managed by its own controller. Thus, a core accessing its own registries is much faster than accessing the registries of another core in the same processor (e.g., 10 vs. 50 clock cycles), which in turn is much faster than going through the memory controller of another CPU in a different socket (e.g., 50 vs. 500 clock cycles).



	Size	Latency	Bandwidth
L1 cache	32 KB	1.5 ns	
L2 cache	256 KB	4 ns	
L3 cache (shared)	30 MB	15 ns	
Main memory (local)	256GB x 2 x 8	60ns-100ns	13-17GB
Main memory (neighbour blade)	4TB	500+ ns	3GB
Solid State memory	Up to TBs	200,000ns	0.5GB
Hard Disk	Up to PBs	10,000,000 ns	0.07GB

Figure 7.2: Memory hierarchy

Fig. 7.2 exemplifies the different levels of memory with their different characteristics (i.e., cost, size, latency[8] and bandwith[9]) of each of them. Concretely, there would be two memory controllers per CPU, and up to eight CPUs per blade, with ten cores each.[10] It is quite easy to appreciate two trade-offs: (i) The smaller the memory, the less latency and more bandwidth it provides; and (ii) The more performant the memory, the more expensive it is.

Besides the benefit of somehow configurable memory access time, it is clear that this hardware architecture also allows parallelism in the different cores and CPUs. However, it may not be that obvious that column-wise storage also makes it easy to execute operations in parallel. We should consider that data in a Column store is somehow vertically partitioned, which means that operations on different columns can be easily processed in parallel. Thus, if multiple columns need to be searched or aggregated, each of these operations can be assigned to a different processor core. This can be further facilitated by implementing some specific virtualization techniques with very limited overhead. For example, when reading memory pages, the hypervisor[11] is often not required to be called.

Such hardware architectures are much faster (and expensive) than a cluster of shared-nothing machines, but it is clear that they are also much harder to manage if we really want to unleash all their power. Therefore, since distance matters (i.e., access time is not uniform), this requires optimizing the use of memory hierarchies to exploit data locality (i.e., move the computation as close as possible to where the data are). But doing this is a really hard problem, which needs to be aware of available resources and requirements everywhere, to maximize the amount of useful data in the more performant memory levels.

There are three important techniques that systems typically use to optimize multi-level cache usage:

Locality Awareness aims at reducing the number of idle CPU cycles waiting for the arrival of data. As explained before, the time to move data somehow depends on the distance from the storage to the processing unit.

[7] `https://en.wikipedia.org/wiki/Non-uniform_memory_access`

[8] Latency in this case corresponds to the amount of time from a request until the data transfer begins.

[9] Bandwidth is the amount of data transferred per second.

[10] Based on Intel Xeon E7-8800.

[11] The hypervisor is the software that coordinates different virtual machines.

Thus, we can benefit from both spacial locality (high probability of accessing contiguous data, specially in case of sequential access), and temporal locality (high probability of accessing the same piece of data several times in a short period of time, which is affected by the cache eviction/replacement policy, typically Least Recently Used).

Flexible Caching has the purpose of bringing and keeping as much relevant data as possible in the cache. This is related to associativity, which is the multiplicity of the mapping between a given storage/cache level and the one above it in the hierarchy. It can be configured to any value, in general, but a typical one is 8 (meaning that, for example, each disk block can only be placed in 8 different memory positions). Low associativity facilitates searching, since a disk block can only be found in few places, but complicates replacement, since it offers few options to choose. Oppositely, high associativity facilitates replacement, but makes it more difficult and slow to find the disk block or know if it is in memory.

Cache-conscious Design is relevant for DBMS developers, that need to be aware of cache line size. The idea is that in the code they only use multiples of that size (padding structures is needed), so that accesses are optimized (never require to access two cache lines for data that could fit in one). This is specially relevant in defining arrays, where we can then easily move from line to line by multiplying the offset.

7.2 Columnar storage

From the perspective of the performance, it is obvious that the objective is to keep as much data as possible in memory (ideally all the database). One way is to buy more memory blades or machines, but another is to reduce the size of data with compression techniques. For this, it is important to realise that compression is easier when data are more similar, and this is specially true when we store them column-wise (e.g., storing intertwined names and ages will require more space than storing all names together and then all ages together, because the later makes data more uniform and consequently more compressible). Some studies indicate that it is easy to achieve a compression factor of ten in Column stores. Thus, we can keep ten times more data in-memory, but they can also be loaded or moved faster (in case they do not completely fit in).

As a side effect of this kind of extreme vertical fragmentation, we maximize the hit ratio on accessing memory (i.e., we only need to access the requested attributes and no others), specially in the presence of projections and aggregations. Obviously, as any vertical fragmentation, it also favours parallelism.

It is also important to remind that one of the advantages of indexing (besides the data structure) is that the index does not contain all the data and is hence smaller than the table. However, there is no reason to occupy memory with auxiliary structures of limited usefulness, when all the effort is being put in reducing the size of data with compression techniques. The column-wise storage together with compression techniques makes the size of the relevant data comparable (or even smaller) to that of an index. Thus, indexes loose most of their relevance in this kind of systems in front of compression (storing data in columns works like having a built-in index for each column).

From the perspective of the use case, the main idea behind the architectural shift is to deal with the dual transactional-decisional workload (a.k.a. HTAP). The former (a.k.a. OLTP) is characterized by many modifications of data, while the latter (a.k.a. OLAP) is characterized by complex queries (including groupings and aggregations) in the absence of modification. Thus, one needs to be frequently persisted in disk, while the other can rely much more in faster memory accesses. Thus, if we want to make compatible both workloads, we also need to allow the storage of tabular data in both ways: row-wise (per entity/instance, all its attributes/features together) and column-wise (per attribute/feature, all the entities/instances together). For example:

A	**B**	**C**	**D**
a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2
a_3	b_3	c_3	d_3
a_4	b_4	c_4	d_4

Row-wise serialization: a_1 ; b_1 ; c_1 ; d_1 | a_2 ; b_2 ; c_2 ; d_2 | a_3 ; b_3 ; c_3 ; d_3 | a_4 ; b_4 ; c_4 ; d_4
Column-wise serialization: a_1 ; a_2 ; a_3 ; a_4 | b_1 ; b_2 ; b_3 ; b_4 | c_1 ; c_2 ; c_3 ; c_4 | d_1 ; d_2 ; d_3 ; d_4

Inserting data row-wise is much faster, because it only requires appending a new record at the end of the corresponding file, while column-wise requires storing new content in different places (e.g., four in the previous example with four columns). On the other hand, selecting some feature and creating an aggregate (e.g., sum, max, min, avg) is much faster column-wise, because all the values are found together and can be efficiently read. Consequently, we have that write-intensive OLTP systems perform better in a disk-based row-wise storage, but read-only OLAP systems perform better in a memory-based column-wise storage (see [AMH08]).

Indeed, this is not new. The conception of an alternative columnar storage of tabular data dates back to the 80s (see [CK85]). However, at that time, the dominant workload was OLTP and consequently all RDBMS chose to store tables row-wise. Nevertheless, as OLAP and DW mostly-read workloads gained relevance, RDBMSs started introducing some mechanisms to imitate to some extent the behaviour and benefits of columnar storage:

Materialized views are views whose content is physically stored in disk. Any view (a.k.a. query) can be materialized, but it comes at the cost (in terms of both space and updating time) of maintaining the consequent redundancy. Thus, we could always define a set of materialized views such that there is a view with exactly the columns needed to answer a query, or with exactly one column each (like in columnar storage). Clearly, the problem would be to maintain these many materialized views.

Index-only query answering allows some DBMSs allow to answer queries only based on the content of indexes (i.e., without accessing the tables). Obviously, the involved indexes must contain all the necessary information to do it (e.g., with an index over an attribute *age*, we can answer the query "`SELECT AVG(age) FROM <Table>;`"). We could thus create a set of indexes that cover all columns used in a query (or table). The problem is that an index does not only contain the attribute values, but also the physical addresses of the records in the table[12], thus, incurring in some overhead.

Vertical partitioning taken to the extreme results in columnar storage. Thus, if our RDBMS allows vertical partitions, we can configure it to generate one partition per column, which would be quite similar to columnar storage, with the important difference, that each vertical partition must have a copy of the primary key of the table, while true column-wise storage relies on the order of tuples instead.

As we see, there are some alternatives that allow to emulate columnar storage, but they always come at a cost. However, this is not the main reason not to take the conservative approach of enriching traditional row-wise RDBMSs. The main reason to implement native columnar storage is the many specific optimization techniques[13] we can implement, as we will see in Section 7.3.4.

Nevertheless, row-wise storage is still very useful in many circumstances. Hence, the idea of the new architecture is to provide both row and columnar engines natively under the same DBMS as depicted in Figure 7.3. The problem that appears is then what to store where (see Section 7.3.1).

Besides the duality of layout, this kind of systems offers other improvements and optimizations. For example, they usually promote the use of stored procedures instead of interactive queries. The reason for this is to improve query optimization and save it in successive calls. Moreover, they keep track of the usage of different rows and implement an aging mechanisms, so that older ones are moved to a separate colder storage system. Modifications are also performed in a differential buffer to avoid in-place updates of rows, which is specially inefficient in case of compression (everything needs to be re-compressed after a single change). Then, when this differential buffer grows beyond some threshold, the table is merged and recompressed again (all this happens on-line). Despite being heavily optimized, it is clear that such merging process is specially expensive. Therefore, tables can be declared to be append-only, which is achieved by adding a timestamp or valid time field to the identifier of every row.

7.3 Data Management

One of the claims of In-memory Column stores is that despite the dual storage they are easier to manage. Indeed, they offer both row-wise and column-wise layouts together, but these are transparent for the user that just poses SQL queries like in any other RDBMS. Moreover, they simplify administration of decisional workloads, because thanks to the column-wise layout, they do neither require managing materialized views nor indexes. Also, ETL is

[12] Indexes store entries, which are pairs [value,address].

[13] According to [SAB+05], column-oriented DBMS implement specific optimization mechanisms that outperform fine-tuned row-oriented RDBMS (up to 50-75% of improvement).

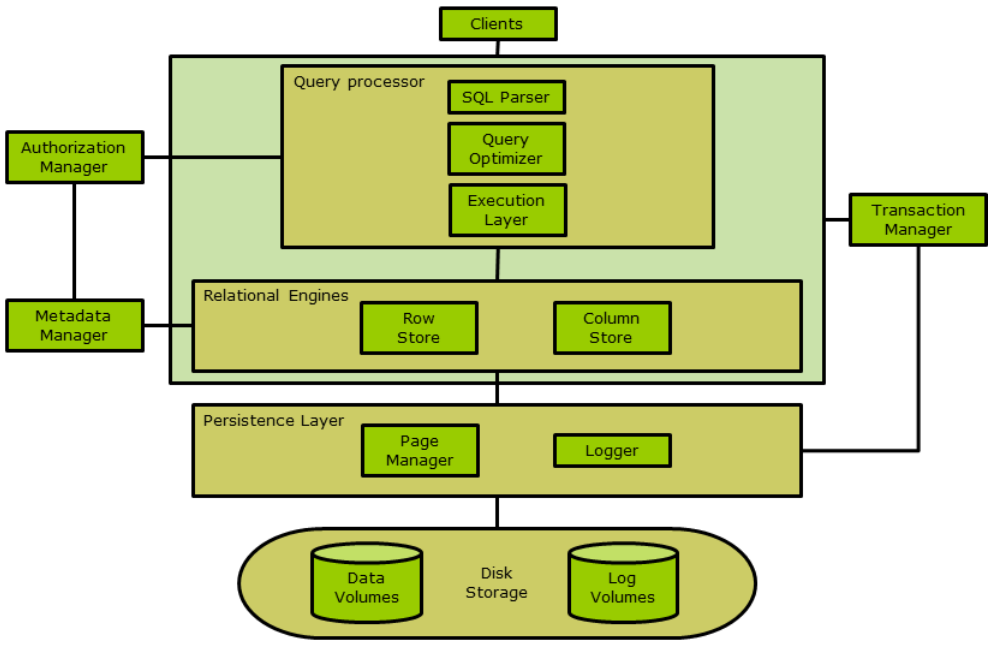


Figure 7.3: SanssouciDB architecture

not needed, because both transactional and decisional queries can be served from the same database. Thus, there is no need to wait for the consequent load of the DW.

7.3.1 Data Design

Managing distribution in this kind of systems is completely transparent to the developers, requiring minimal configuration from the DBA. Nevertheless, still some things need to be considered.

7.3.1.1 Fragmentation

In general, deciding the best layout being more row-wise or column-wise (i.e., which attributes are stored together) for a set of tables is an NP-hard problem, equivalent to generic vertical fragmentation[14] (see Section 3.1.2.1.2). It requires to know the workload (including weight and cost of each query) and then analyze the cache miss behaviour. Moreover, besides being computationally expensive and inherently difficult, current techniques for fragmenting a database vertically, such as attribute clustering or affinity matrix (see [ÖV20]), do not consider evolution and assume only static workloads. Thus, given the cost of finding the optimal solution, an acceptable alternative is to use heuristics.

7.3.1.1.1 When to use column-wise storage Column-wise storage is heavily penalized in front of insertions, but also has many advantageous situations:

1. When calculations are executed on a single column or a few columns only. By reading only the data relevant to answer the query (and not those irrelevant columns), you increase the hit ratio[15].
2. When the table is searched based on the values of few columns. If the query is really selective (i.e., it accesses few rows) and the accessed data is decided on the values of few columns (ideally one), we can process first the relevant column, and only access the others for the few rows fulfilling the predicate.
3. When the table has a large number of columns. Tables with many columns are rarely fully accessed (usually only some of the columns are retrieved).

[14] Notice that having a fragment with one single attribute is equivalent to store its data column-wise.
[15] In this context, the hit ratio is the percentage of relevant data you find in a block.

4. When the table has a large number of rows, and columnar operations are required (i.e., aggregate, scan, and so on). The bigger the table, the bigger the absolute gain (in both compression and saving accesses to irrelevant data).

5. When the majority of columns contain only a few distinct values (compared to the number of rows). The more repetitions, the higher compression ratio we get.

7.3.1.1.2 When to use row-wise storage As we said, column-wise storage is not always good. One of the problems it generates is that of inserting new rows, but also there are some situations in which it simply does not generate any benefit and complicates management:

1. When the table has a small number of rows (e.g., configuration tables). If all of it fits easily in memory without any kind of compression, columnar storage would be an overkilling solution. Moreover, the overhead incurred when managing fragments cancels out the performance benefits of exploiting parallelism.

2. When the application needs to process (i.e., select or update) only an attribute of a single record at a time, typically accessed by primary key. Accessing one datum results in one I/O irrespectively of the layout being used.

3. When the application typically needs to access the complete record. Columnar storage would spread the data in different blocks, generating many I/Os when there could be only one if stored altogether.

4. When the columns contain mainly distinct values so the compression ratio would be low. It is clear that the more redundancy the more compression, and vice-versa. Thus, if we do not have redundancies, there is one less reason to store the data column-wise.

5. When aggregations and fast searching are not required. If we said that this kind of access benefits from columnar storage and we do not have it, we also lose a reason to use that.

7.3.1.2 Allocation

Consider a workload consisting in a set of statements/queries that access existing fragments, and whose operations consume resources (i.e., CPU and memory) of limited capacity in the site where they are executed. Moreover, we need to consider also that these systems usually use high speed (e.g., $10GB/s$) network links for communication among nodes. Nevertheless, transferring data in fast network links like these can still make execution times significantly slower than they would be if all required data was available in a single host (notice that we are comparing against in-memory computation). Besides, each fragment has certain characteristics such as number of rows and size that influence the cost of operations.

Thus, our problem is the assignment of data fragments to specific locations in the system, with the goal of maximizing performance. This implies that none of the nodes is overloaded with more fragments or more operations than it can handle and the data transfer in the network is minimized. This is clearly a Combinatorial Optimization Problem aiming at making a set of decisions (i.e., allocate each fragment) in a way that maximizes an outcome (i.e., performance). It is well known that the decision versions of Combinatorial Optimization Problems are NP-Complete. Consequently, this is an open research problem and only heuristics can be used by now (e.g., take some greedy approach and place together all fragments being used in the most critical queries).

7.3.1.3 Replication

As already stated, replication creates the problem of synchronizing replicas, but we should not forget that it also increases the amount of space used. This is typically not relevant when we talk about disk storage, but results crucial in the case of in-memory DBMS. Thus, the approach these systems take is to do it only for small tables that are rarely updated. An example of such tables are dimension tables in star-join multidimensional schemas.

7.3.2 Catalog Management

There is not much specificity in the management of the catalog. As we can see in Figure 7.3, it is similar to that of any RDBMS. The only thing worth to mention is that its tables are not stored column-wise, but row-wise. Rationale for this can be seen in Section 7.3.1.

7.3.3 Transaction Management

One of the main problems in analytical architectures is the propagation of changes in the OLTP environment to the OLAP environment. The application needs to immediately update the aggregated value after a basic item was either added, deleted or modified. Alternatively, special aggregation runs need to be scheduled that update the aggregated values at certain time intervals (for example once a day). This is typically done with either materialized view updates or more complex ETL processes. Either one way or another, it is clear that we are talking about some kind of data redundancy that we can interpret in terms of replication management.

In the case of in-memory column stores, this redundancy is avoided and only atomic data is stored, completely relying on the excellent performance obtained in aggregate queries. This way, the application logic is clearly simplified. Indeed, by eliminating persisted aggregates, the additional logic to propagate changes is no longer required. With on-the fly aggregation, the aggregate values are always up-to-date, while materialized aggregates in classical RDBMS (in the form of materialized views) are sometimes updated only at scheduled times. Thus, we always get contemporaneousness of aggregated values.

As a side effect, we also reach a higher level of concurrency. With materialized aggregates after each write operation a lock needs to be acquired for updating also the aggregate, which limits concurrency and may lead to performance issues. If we do not materialize aggregates only the basic items are written, and this can be done concurrently with the finest granularity lock.

Nevertheless, as explained above, we can still replicate some small and mostly static tables. Unlike NOSQL, this kind of systems provide full support for ACID transactions. Thus, in the unlikely case that these change, the modification is done synchronously, which means that given the strong consistency model they follow, it does not make much sense to make a distinction between primary or secondary copy. Therefore, we might say that they implement an Eager/Seconday-copy synchronization mechanism.

7.3.4 Query Processing

First of all, it is important to highlight that these systems take the full power of any kind of parallelism explained in Section 3.1.5.4, which is somehow easier in a shared-memory architecture than in shared-nothing. Nevertheless, this is still far from trivial. On the one hand, fragmenting presents the added difficulty of being always hybrid (i.e., both horizontal and vertical), because of the duality of storage. On the other hand, pipelining (that favours inter-operator parallelism) is specially hard because of the lack of height of process trees, synchronization barriers and skewed cost of operators (e.g., selections require very short time, while joins need much longer execution). Moreover, some specificities of the architecture also penalize parallelism. Firsly, the many cores potentially used in a single operator have to be initialized, which consumes many resources. Then, their communication generates hardware contention due to the shared communication chanels and memory controllers. Finally, the predominant small size of operators in the process trees maximizes the impact of skewness in their execution time (i.e., little variations in execution time are more relevant for operators with smaller average execution time in their parallel tasks).

Besides the architectural benefits of parallelism, these Column stores boost performance by implementing some specific query processing optimizations that cannot be done in general (definitely not by traditional row-oriented RDBMS with some specific mechanisms, i.e., vertical partitioning, index only query answering or materialized views).

- Late materialization delays as much as possible joining back all the columns and allows to process the query without reconstructing the original table. Obviously, this saves memory space as well, but also allows to parallelize the processing of different columns.

- Tuples are identified by their position. Oppositely, in row-wise storage, a tuple can be in any disk position and must always be accompanied by the primary key. Thus, if we compare column-wise storage against vertical partitioning in row-wise storage, the difference is that in the former, the primary key does not need to be replicated with each column (this saves both space and time in the queries).

- Column-specific compression techniques are applied (see Section 7.3.4.1). Compression can be applied in any RDBMS, but techniques differ. Some reduce more the space but are computationally heavier. In the case of Column stores, the most important point it not to reduce the space as much as possible, but to be able to

process the queries without decompressing the data (otherwise, we would occupy the memory anyway and compression would be mostly useless).

- Big homogeneous regions are more compressible than blended heterogeneous ones. Thus, sorting data appropriately can improve compression. The problem is that what favours the compression of one column penalizes another one, but since column values are identified by position, all columns must follow the same order. Thus, a potential optimization technique in Column stores is replicating the table and use a different row order in each of the replicas. Then, every query will use the most beneficial order, depending on the columns it requires. Obviously, this would affect replica synchronization as explained in Section 7.3.3.

- During query processing in row-wise storage, data are iterated per row. It is easy to see that this contradicts the goal of working with data still compressed and late materialization, because both delay the reconstruction of tuples. Instead, Column stores process data per blocks, where one block corresponds to some subset of a compressed column. This is known as block iteration, and in combination with late materialization is known as vectorized query processing. Thus, operations on single columns, such as searching or aggregations, can be implemented as loops over an array stored in contiguous memory locations, which has high spatial locality and can be efficiently executed in the CPU cache.

- It is easy to imagine that with compressed data and late materialization, we will need some specific join algorithms, as well.

7.3.4.1 Compression

It is clear that data stored in columns is more compressible than data stored in rows, because they show high data value locality (i.e., closer values are more similar and have less entropy), but may not be that obvious that the main objective of compression in Column stores is not to reduce the space in the disk, but the I/O and the amount of memory used in processing queries. Thus, heavyweight compression (e.g., Lempel-Ziv), which is good if there is a (huge) gap between memory bandwidth and CPU performance, may not be that useful in this case. Alternatively, it is more promising to rely on lightweight compression (e.g., Dictionary or Run-Length encoding) that in general will not reduce the size of data that much, but may allow the query engine to work directly on compressed data (e.g., decompression is not needed to evaluate complex predicates with conjunctions and disjunctions). In general, compression can speed up operations such as scans and aggregations if the operator is aware of the compression.

[PZ11] describes some simple and popular lightweight compression techniques:

Dictionary encoding. Table columns which contain only a comparably small number of distinct values can be effectively compressed by enumerating the distinct values and storing only their numbers. This technique requires that an additional table, the dictionary, is maintained which in the first column contains the original values and in the second one the numbers representing the values. This technique is very common and leads to high compression rates (e.g., in country codes or customer numbers), but is seldom regarded properly as a compression technique.

Common value suppression. In many cases, there is a predominant value in a column (typically NULL value in RDBMS). In these cases, a bit map can be built, so that the bit is set if the corresponding row contains such value. Then, only when the bit is not set, we really need to store/retrieve something.

Cluster encoding. This compression technique works by searching for multiple occurrences of the same sequence of values within the original column. The compressed column consists of a two-column list with the first column containing the elements of a particular sequence and the second column containing the row numbers where the sequence starts in the original column. Many popular data compression programs use this technique to compress text files.

Run-length encoding. If data in a column is sorted there is a high probability that two or more consecutive elements contain the same values. Run-length encoding counts the number of consecutive column elements with the same values. To achieve this, the original column is replaced with a two-column list. The first column contains the values as they appear in the original table column and the second column contains the counts of consecutive occurrences of the respective value. From this information the original column can easily be reconstructed.

Bit compression. Instead of storing each value using a built-in integer data type, bit compression uses only the amount of bits actually needed for representing the values of a sequence. That is, the values are stored using fixed-width chunks, whose size is given implicitly by the largest value to be coded.

Variable byte coding. Compared to bit compression, variable byte coding uses bytes rather than bits as the basic unit for building memory chunks. Generally, all values are separated into pieces of seven bits each, and each piece is stored in a byte. The remaining leading bit is used to indicate whether this is the last piece of a value. So, each value can span a different number of bytes. Because the memory of almost all modern computer systems is byte-addressable, compressing and decompressing variable byte codes is very efficient. It is also possible to generalize variable byte coding using chunks with sizes other than eight bits.

Chapter 8

Distributed Processing Frameworks

In this chapter, we review the most well-known distributed data processing frameworks, that allow to run tasks in parallel. As previously said, this is nothing new. Parallel architectures were already introduced in the 90s (see [DG92]). However, it became specially trendy thanks to Google, who published its MapReduce framework in [DG04]. It immediately became mainstream as an Apache project[1]. Nevertheless, as happens with many NOSQL tools, it was too specific and lacked generality to be useful to everybody. Thus, the University of California, Berkeley started a project and developed a generalization as part of a PhD thesis (see [Zah13]), which became open source in 2010 and moved to the Apache Software Foundation in 2013[2]. In 2015, it was already one of the most active open source Big Data projects. Both frameworks share the same objective of minimizing data transfer (hence, maximizing processing local to the data). To that end, they widely benefit of distributed storage systems such as Apache Hadoop.

8.1 MapReduce

The MapReduce framework was the pioneer on massively distributed data processing, and is the default engine in the Hadoop ecosystem. It was originally developed at Google by Jeffrey Dean and Sanjay Ghemawat (see [DG04]). Prior to it, specific programs had to be implemented for data crawling, log analysis, etc. Yet, programmers faced the difficulty to distribute and parallelize their code in order to be efficiently executed over massive datasets. In order to overcome these complex issues, MapReduce was developed to offer programmers an abstraction that allowed them to specify their implementations in terms of two functions, `map` and `reduce`. Execution details like parallelism and distribution were automatically managed by the underlying framework, which facilitated growing to the scale of the Cloud. This creates the extra problem of fault tolerance. Give the high number of machines involved in a long processing, it was quite likely the some of them crashed in the meantime. Therefore, restart should be avoided and, in case some machine fails, some other should automatically take over.

8.1.1 Programming model

MapReduce paradigm is inspired by the *map* and *reduce* primitives available in the LISP functional programming language[3]. In LISP, the *map* function takes as argument a unary function and a set of values, such that the function is applied to each of the values. For instance, the following statement

(**map** $2\times$ (3,4,5,6))

doubles each of the values in the provided set, hence yielding the set $(6, 8, 10, 12)$. Then, the *reduce* function takes as argument a binary function and a set of values, which are pair-wise combined using that function. For instance, the following statement

(**reduce** + (6,8,10,12))

[1] `http://hadoop.apache.org`
[2] `http://spark.apache.org`
[3] `https://lisp-lang.org`

has as input the output of the previous *map* and sums all its elements, hence yielding the value 36. Notice that the *map* operator can be executed in parallel over each of the elements, since there is no dependency among them. Not that easily, but the *reduce* operator can also be parallelized.

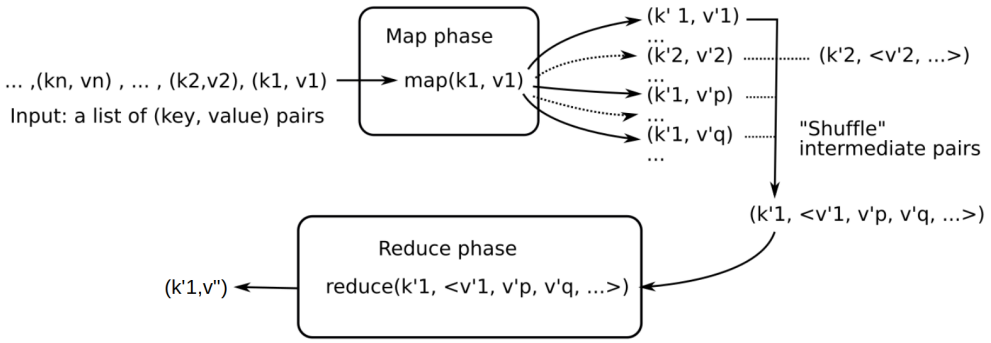


Figure 8.1: A high-level overview of the MapReduce programming model, from [AMR+11]

MapReduce deals with key-value pairs, which is the minimum unit of processing. Intuitively, given a set of input pairs, the `map` function is applied to each of them. This generates a set of intermediate key-value pairs, where both the key and value can be modified with respect to the input. Then, each intermediate pair is *shuffled* so that all equal keys meet together in the same machine. Thus, a set of values is associated to each distinct key (i.e., all those values that were mapped to that key). These serve as input to the `reduce` function, which can generate another set of key-value pairs for each distinct key. Figure 8.1 provides a high-level overview of the MapReduce programming model.

A formal definition of the programming model is as follows. Let $I \subseteq T_{IK} \times T_{IV}$ be the set of all possible input key-values, where T_{IK} is the domain of input keys and T_{IV} the domain of input values. Then, MapReduce defines the `map` as a function $I \rightarrow 2^M$, where $M \subseteq T_{MK} \times T_{MV}$ (i.e., the domain of map keys and map values). The output of the map function is shuffled so equal keys are grouped together. Then, the `reduce` function is formally defined as $T_{MK} \times 2^{T_{MV}} \rightarrow 2^R$, where $R \subseteq T_{RK} \times T_{RV}$ (i.e., the domain of reduce keys and reduce values).

8.1.1.1 Phases of a MapReduce job

We, now, describe the phases that a MapReduce job undergoes:

i) Input. Reads data from a DFS. If the input file format is already fragmented into a key-value structure, such as SequenceFile, the key and value are taken from the file. In the case the input file is raw (e.g., a CSV file), then the framework automatically constructs a key-value structure, where the key is the tuple offset in the file and the value the row itself.

ii) Map. For each input key-value pair, the Mapper machines execute the user-provided `map` function, which can return 0-to-many new pairs.

iii) Partition. The generated map key-value pairs are assigned to Reducer machines based on the key. This phase guarantees that all occurrences of the same key will be assigned to the same Reducer. Note, however, that data are not shipped yet in this phase.

iv) Shuffle. The Reducer machines pull the generated key-value pairs from the Mapper machines (those that have been defined in the previous phase).

v) Sort&Comparison. Each Reducer machine sorts their input key-value pairs based on the key. This allows to generate, for each distinct key, their corresponding set of values via the merge-sort algorithm.

vi) Reduce. For each input structure key-set of values, the `reduce` function is executed, which can return 0-to-many key-value pairs.

vii) Output. The result of the `reduce` function is written locally at each Reducer machine leveraging on the DFS.

8.1.1.2 Examples

Now, we can see a simple example of MapReduce to count words in a text, to see later the implementation of two generic Relational operators (namely Projection and Cartesian Product).

8.1.1.2.1 Word count The most well-known MapReduce example is that of counting words. Let us assume a DFS, containing a bunch of files with text books in it. The task here is to aggregate, for each distinct word, all its occurrences across all books.

The `map` task is assumed to be invoked with key as the file path, and value the actual content of the file. Then, the function iterates for each word in the document and emits the pair $(w, 1)$. This indicates that, each word w has been seen once.

```
function map(String name, String document) {
  for w in document:
    emit(w,1)
}
```

The `reduce` function receives, for each word, a set of integers (allegedly 1's) indicating how many times it has been seen. Then, the task here consists on summing all of them in order to provide a global aggregate.

```
function reduce(String word, Iterator counts):
  sum = 0
  for c in counts:
    sum += c
  emit(word, sum)
```

Note that, here, a `combine` function would be useful and coincident with the `reduce` one, as the sum of natural numbers is commutative and associative (see Section 8.1.2.1.1 for the description of `combine`).

8.1.1.2.2 Relational algebra operators The relational algebra operators can also be distributed and parallelized with MapReduce. Here, we provide the examples of the projection and cartesian product. We assume that for an input $[k, v]$, the operator $k \oplus v$ is the reconstruction of the whole tuple (the key, for instance being a primary key or a row identifier).

Projection The bulk of the work to project a set of attributes $a_{i_1}, \ldots, a_{i_n}$ is done at the `map` function. Here, the projected attributes are returned as the key, the value in the output being irrelevant for the function. Then, the `reduce` function emits the tuple. Note that, this version of the projection operator assumes *set semantics* (i.e., there are no duplicate tuples in the result). It is left out to the reader how the projection should be adapted in order to assume *bag semantics* (i.e., every tuple can appear more than once in the result).

$$\pi_{a_{i_1},\ldots,a_{i_n}}(T) \Mapsto \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto [(\texttt{prj}_{a_{i_1},\ldots,a_{i_n}}(k \oplus v), 1)] \\ \texttt{reduce}(\text{key } ik, \text{value } ivs) \mapsto [(ik)] \end{cases}$$

Cartesian Product The `map` function applies a different logic depending on whether the input tuple belongs to T (i.e., the *external* Relation) or S (i.e., the *internal* Relation). If the tuple corresponds to T, we generate a key with a hash function modulo D (where D should be the desired number of calls to the `reduce` function) and emit it. Otherwise, if the value corresponds to S, we emit the same tuple in the value paired with all keys in the range $0, \ldots, D-1$. Then, the `reduce` function receives the mixed tuples belonging to T and S. Then, it classifies them into two sets, and performs the cartesian product of these sets.

$$T \times S \Mapsto \begin{cases} \texttt{map}(\text{key } k, \text{value } v) \mapsto \begin{cases} [(h_T(k) \bmod D, k \oplus v)] & \text{if input}(k \oplus v) = T \\ [(0, k \oplus v), \ldots, (D-1, k \oplus v)] & \text{if input}(k \oplus v) = S \end{cases} \\ \texttt{reduce}(\text{key } ik, \text{value } ivs) \mapsto [\text{crossproduct}(T, S) | T = \{iv | iv \in ivs \wedge input(iv) = T\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad S = \{iv | iv \in ivs \wedge input(iv) = S\}] \end{cases}$$

8.1.1.3 Characteristics

On the positive side, MapReduce is not limited to SequenceFiles, not even to HDFS files. Indeed, we can find implementations for HBase and MongoDB among others. Moreover, we have to highlight that as long as `reduce` produces key-value pairs, its signature is closed, which means that we can chain different MapReduce jobs. However, since these are independent processes, fault tolerance is not guaranteed between them, and resources are realeased to be just immediately and inefficiently requested again.

On the other hand, the MapReduce programming paradigm is computationally complete, in the sense that any data process can be adapted to it. Nevertheless, the main criticism it receives is that of being too low level (specially compared to a declarative language like SQL). In particular, any SQL query can be implemented in MapReduce, but this does not mean that it results in the most efficient execution. One of the main problems is that optimization is compromised because of its lack of expressivity (only `map` and `reduce` operations). Thus, different projects offering SQL-like syntax on top of MapReduce (e.g., HiveQL or Cassandra Query Language) fail to provide a good query optimizer.

8.1.2 Anatomy of a Hadoop MapReduce job

In the following, we will explain how this paradigm is implemented leveraging on the distributed storage provided by HDFS. We will also present the technical details in the MapReduce framework, which is designed with the main assumption that the data elements (i.e., tuples, documents, etc.) can be processed independently. This allows to maximize parallelism, as the processing of a large dataset can be spread across several processing units.

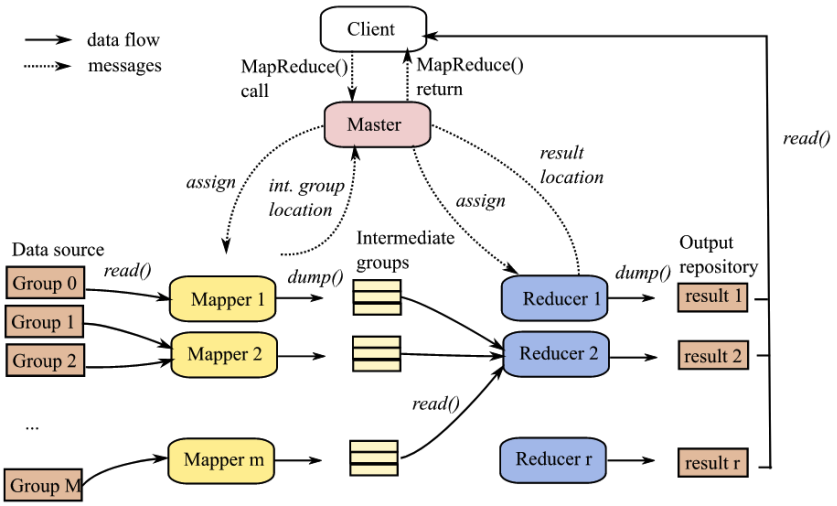


Figure 8.2: The execution of a distributed MapReduce job, from [AMR+11]

As depicted in Figure 8.2, the distributed execution of a MapReduce job is performed in *Mapper* and *Reducer* server nodes, which are responsible of executing multiple `map` and `reduce` calls each. For each input key-value pair a `map` call will be performed, executing the user-provided function. The same mechanism also applies to distinct keys to `reduce` calls. In order to fully benefit from data locality, the MapReduce framework is meant to be executed in a DFS such as Apache Hadoop. Indeed, the SequenceFile format already implements key-value structures, being a natural fit to serve as input to MapReduce jobs. Hence Mappers should be running in the same servers hosting HDFS DataNodes.

Thus, the execution flow is as follows. Firstly, (1) the user program forks and creates the coordinator (a.k.a. master) and worker nodes (M mappers and R reducers). Then, (2) these are assigned different map and reduce tasks in independent JVM. Each of the mapper is (3) assigned a piece of the input (a.k.a. split), usually sitting in the same cluster node (benefitting from data locality). After processing all the input, each mapper (4) writes its results to the DFS in one different file per reducer (notice that this means $M \cdot R$ intermediate files). When all mappers have finished, then reducers are launched and (5) each of them remotely reads its corresponding intermediate files in HDFS. Finally, after finishing its task (6) each reducer writes its results to the disk. Notice that during all

this process, the master node keeps track of the status of the different tasks (periodically pings workers) and the files being generated. It is thus the master that indicates to each reducer the files in HDFS that it has to read and process. In case some worker (either mapper or reducer) is not responding, the corresponding task is immediately reassigned (notice that all files are replicated in HDFS). The failure of the master is more unlikely, but to mitigate the effect this would happen, we can create checkpoints of its in-memory structure tracking the status of tasks and intermediate files.

One of the key issues in this process is to configure the number of mappers and reducers, which in the end are an indicator of the level of parallelism we can achieve. The number of map tasks depends on the splitting of the input, which is by default performed per HDFS chunk. As an indicator, we can consider that each node should be running between ten and hundred mappers, each taking more than one minute to execute (for their creation overhead to be worth). On the other hand, there are two different configurations suggested for reducers. One minimizes the number of tasks (and hence the number of intermediate files) by creating slightly less reducers that the overall number of processors available. The other option, that aims to balance the workload, is to create less than double reducers compared to the available processors.

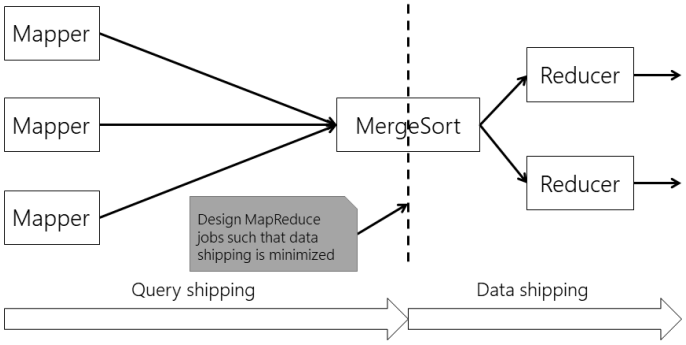


Figure 8.3: MapReduce phases

8.1.2.1 Hadoop MapReduce execution steps

At this point, it is clear that the MapReduce execution is happening in two phases separated by a massive data movement during the sorting, sketched in Figure 8.3. Thus, we can see it as if during the first phase, processing is moved where the data is (i.e., query shipping). Then, during the second phase, data is moved where processing must happen (i.e., data shipping). In between, part of the sorting happens actually before moving the data and some after moving the data. Concretely, the step by step execution of MapReduce is as follows:

1. Upload the data to the Cloud
   - This partitions them into blocks
   - Originally, using HDFS, but currently there are alternatives (e.g., HBase, MongoDB, Cassandra, CouchDB, etc.)
2. Replicate them in different nodes
3. Each mapper (i.e., JVM) reads a subset of disk blocks (a.k.a. split)
4. Divide each split into records
5. Execute the `map` function for each key-value pair and keep its results in memory
   - JVM heap is used as a circular buffer
6. Each time memory becomes full
   (a) Partition memory per reducers
      - Using a hash function f over the new key generated in the output of the `map` function, modulo the number of reducers
   (b) Sort each partition independently

(c) Spill partitions into disk
    - Massive writing
    - Releases memory
7. Partitions of the same reducer produced in different spills are sorted and merged
    - Each merge is processed independently (i.e., partitions are not undone)
8. Store the result into disk
9. Reducers fetch data through the network
    - Massive data transfer
10. key-value pairs are sorted and merged in the reducers
11. `reduce` function is executed per key
12. Store the result into disk

8.1.2.1.1 The `combine` function The avid reader will have realized that there is a possible optimization in the previously described steps. Precisely, it is possible that many pairs with the same key are being generated in the same Mapper. Hence, in some cases, a local `reduce` can minimize the amount of pairs stored in disk and shuffled over the network. Indeed, during Step 6b and 7, if a `combine` function is defined, it is executed locally while sorting and merging.

Such `combine` function is commonly the same as the `reduce`, although not necessary. In order to have the same `combine` and `reduce` functions, it is required for the `reduce` to be commutative and associative. For example, the sum, minimum and maximum are both commutative and associative. However, counting and the average are commutative, but not associative. This does not mean that we cannot compute the average but just that the `combine` implementation cannot coincide with that of the `reduce`.

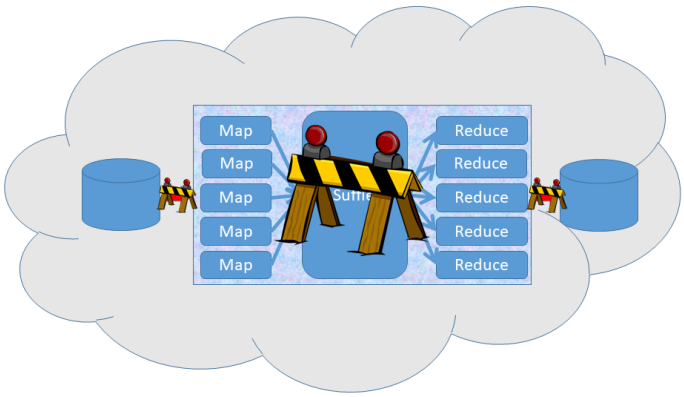


Figure 8.4: MapReduce synchronization barriers

8.1.2.1.2 Problems and limitations In this process, there are basically three synchronization barriers depicted in Figure 8.4. Firstly, nothing can start until all data has been uploaded to the Cloud. Then, all mappers need to finish before reducers can start processing the data. Finally, in case of chaining different MapReduce jobs, the second one cannot start until the first one finishes the writing. Besides these barriers, there are also some painpoints and limitations of MapReduce framework that we need to remember:

- Startup time is very high, because it requires starting multiple JVM in different nodes of the cluster of machines.
- The master is actually a single point of failure.
- Even if not critical, reassigning tasks to the workers in case of failure is also expensive and requires redefining the execution plan on the fly, scheduling (and keeping track of) chunks one by one.

- In general, tasks are assigned locally where the chunks are, but if they are not evenly distributed in the cluster (not even considering replicas), some of the chunks need to be moved to the available processors.
- Intermediate results are written to disk for fault tolerance, but it has a cost.
- Reducers fetch all the data from remote nodes.
- Even if data is compressed in the disk, it will be decompressed before processing (as opposed to what Column stores do).

8.2 Spark

The Apache Spark[4] framework is a natural evolution and generalization of MapReduce. Spark was originally developed at the University of California, Berkeley's AMPLab in the context of Matei Zaharia's PhD thesis [Zah13], and is nowadays one of the major projects in the Apache Software Foundation.

8.2.1 Addressed limitations

Spark is a novel distributed data processing framework which aims to address some major limitations posed by MapReduce.

Composition. Real-world workloads consist of data pipelines encompassing multiple MapReduce tasks, where the output of one is the input of the following one. Yet, to the MapReduce framework this is treated as multiple independent jobs, and thus incur a major cost to start them up.

I/O latency in disk. MapReduce exchanges data between tasks and guarantees fault tolerance by writing into the underlying disk (i.e., a DFS). This has a major impact on performance, as the I/O latency on disk is orders of magnitude slower than that of main memory. Spark aims to minimize this by in-memory pipelining data (locally) when possible.

Limited scope. The MapReduce framework is limited to the semantics of the `map` and `reduce` functions. However, there exist other kind of workloads that would benefit from more simpler or complex kind of operations with other semantics.

8.2.2 Resilient Distributed Datasets

Spark relies on Resilient Distributed Datasets (RDDs) as data structure that allows distributing the underlying dataset with fault-tolerance guarantees. In [Zah13], the following definition of RDD is provided:

> *Unified abstraction for cluster computing, consisting in a read-only, partitioned collection of records. Can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs.* –(Matei Zaharia)

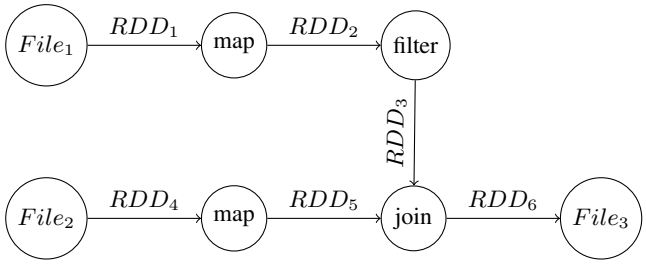


Figure 8.5: Example of Spark job as a DAG

[4] `https://spark.apache.org`

As previously defined, an RDD can be hence only created by reading from disk or a database (e.g., from a Parquet file stored in HDFS), or be generated from another RDD. This entails that RDD cannot be updated, and a complex pipeline will always entail the definition of multiple RDD. In practice, RDD do not hold data, but rather contain the necessary information on how to transform a flow of data passing through them and where to send them. A Spark job is thus defined by a set of chained operations that generate RDD in a directed acyclic graph (DAG) structure. Figure 8.5, depicts an example of a Spark job that independently reads two files and applies certain transformations (i.e., map and filter). Vertices are operations and the edges show how RDD are produced/consumed by them. Thus, the two independent sources are joined into a single RDD in order to write its contents into an output file.

We can summarize the differences between MapReduce and Spark in that (1) MapReduce works on key-value pairs, while Spark can process records with any structure; (2) Results of any MapReduce operations are always persisted in disk form fault tolerance, while Spark tries to save disk writes as much as possible; (3) MapReduce dataset partitions are statically defined by framework configuration parameters, while in Spark this is dynamically defined by parametrization of function calls; and (4) MapReduce only offers two operation (or three if we consider `combine`), while Spark offers many more operations including some binary. In the following sections, we detail the kind of RDD and operations that exist to generate them.

8.2.2.1 Transformations and Actions

Spark distinguishes two kinds of operations: *transformations* and *actions*. A transformation is an operation that defines a new RDD from another RDD (or disk data) without immediately computing it (i.e., they are run *lazily*). An action triggers the computation of dependant transformations, and ultimately returns a value to the program or writes data to a storage system (e.g., a DFS). We can consider that transformations are the internal vertices in the DAG of operations, while actions are the sink vertices. There exist several transformations and actions that implement much more fine grained semantics than MapReduce. The following list exemplifies some of the commonly used transformations[5]. It is important to notice that some operations work on Base RDD containing generic rows of any type (i.e., $RDD[T]$), while others require a minimum structure distinguishing the key from the value and work on Pair RDD (i.e., $RDD[(K, V)]$). One RDD can always generate one from the other but depending on the language this is explicit (e.g., Java) or implicit (e.g., Python).

- $map(f : \text{T} \Rightarrow \text{U}) : \text{RDD[T]} \Rightarrow \text{RDD[U]}$
- $filter(f : \text{T} \Rightarrow \text{Bool}) : \text{RDD[T]} \Rightarrow \text{RDD[T]}$
- $flatMap(f : \text{T} \Rightarrow \text{Seq[U]}) : \text{RDD[T]} \Rightarrow \text{RDD[T]}$
- $sample(fraction : \text{Float}) : \text{RDD[T]} \Rightarrow \text{RDD[T]}$ (with deterministic sampling)
- $groupByKey() : \text{RDD[(K,V)]} \Rightarrow \text{RDD[K, Seq[V])]}$
- $mapToPair(f : \text{T} \Rightarrow \text{(K,V)}) : \text{RDD[T]} \Rightarrow \text{RDD[(K,V)]}$
- $reduceByKey(f : \text{(V,V)} \Rightarrow \text{V}) : \text{RDD[(K,V)]} \Rightarrow \text{RDD[(K,V)]}$
- $union() : \text{(RDD[T],RDD[T])} \Rightarrow \text{RDD[T]}$
- $join() : \text{(RDD[(K,V)],RDD[(K,W)])} \Rightarrow \text{RDD[(K,(V,W))]}$
- $cogroup() : \text{(RDD[(K,V)],RDD[(K,W)])} \Rightarrow \text{RDD[(K,(Seq[V],Seq[W]))]}$
- $mapValues(f : \text{V} \Rightarrow \text{W}) : \text{RDD[(K,V)]} \Rightarrow \text{RDD[(K,W)]}$ (preserves partitioning based on K)
- $sort(c : \text{Comparator[T]}) : \text{RDD[T]} \Rightarrow \text{RDD[T]}$

Additionally, Spark has available the *cache* and *persist* transformations, which have a special treatment. Any of these transformations forces the system to persist the intermediate results corresponding to the RDD, which can be useful to reuse in the case an operation has multiple outgoing edges in the lineage graph. Nevertheless, it is

[5] $RDD[T]$ (resp. $Seq[T]$) denotes an RDD (resp. sequence) of elements of type T.

important to keep in mind that they are only transformations and consequently do not trigger any real execution (only when a descendant action is called). Data can be persisted either in memory or disk, but are evicted from memory if not enough space is available. The eviction policy, applied per partition, is Least Recently Used (LRU).

Regarding actions, the following list provides some of the actions available in Spark:

- $count()$: RDD[T] $\Rightarrow$ Long
- $collect()$: RDD[T] $\Rightarrow$ Seq[T]
- $reduce(f : (T,T) \Rightarrow T)$: RDD[T] $\Rightarrow$ T
- $lookup(k : K)$: RDD[(K,V)] $\Rightarrow$ Seq[V]
- $save(path : String)$: Outputs the RDD to external storage (e.g., HDFS)

Transformations and actions that take as argument a function f expect the programmer to provide its internal logic. Hence, for instance, the `map` function will deal with the transformation of a record (e.g., split a CSV line and extract one attribute), while the `filter` function will implement a predicate deciding whether each particular record is sent or not to the final output.

Example. Let us assume a dataset containing, for each line, a word. The following code, which implements the word count in a MapReduce fashion, would store in a file the number of occurrences of each item.

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaRDD<String> words = textFile.flatMap(s -> {
   return Arrays.asList(s.split(" "))
});
JavaPairRDD<String, Integer> pairs = words.mapToPair(s -> {
   return new Tuple2<String, Integer>(s, 1);
});
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(a,b -> {
   return a + b;
});
counts.saveAsTextFile("hdfs://...");
```

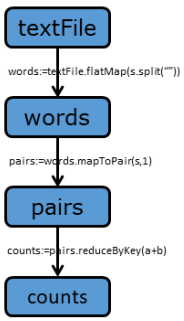


Figure 8.6: Java code for the word count example with Spark

If the dataset, however, contained multiple words per line it, would suffice to add a `flatMap`, in order to generate more than one key-value pair per input line. Figure 8.6 shows the Java code implementing this. To the RHS, there is a diagram showing the four RDDs in green boxes, together with the transformations that generate each. Firsly, the file is read and splitted in to lines. Then, a flatmap divides lines into words, that are mapped into pairs having the word itself as key and a constant one as value. Finally, those pairs are reduced by keys adding the values pair-wise, which eventually generates the overall count of each word.

8.2.3 Spark Under the Hood

In the next sections, we are going to see how Spark is internally working.

8.2.3.1 System architecture

Just like Hadoop and MapReduce, Spark follows an architecture with a single coordinator. Figure 8.7 provides a high-level overview of an Spark cluster. The coordinator is named *driver*, while the rest of machines are the *workers*, which are in charge of executing the data pipeline.

The driver is in charge of creating the context, which is the main entry point for applications to create RDDs. Then, the driver generates a DAG structure like that in Figure 8.5, as well as the internal structures that allow to translate user-defined code into a set of tasks. Spark also provides a built-in resource manager. This is a software

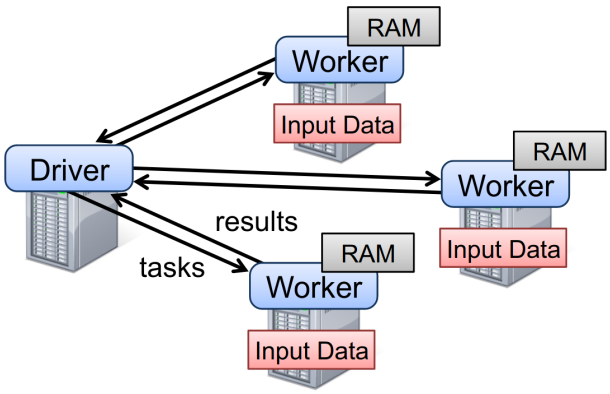


Figure 8.7: Architecture of a Spark cluster

module that arbitrates resources among the different applications of a system. In the context of Spark, hence, the resource manager allows to run jobs in parallel, schedules assigning resources for their execution (e.g., disk, RAM, etc.). However, external resource managers, such as Mesos[6], YARN[7], or Kubernetes[8] can also be plugged to Spark and replace the built-in resource manager.

Spark tries to work exclusively in memory (except for shuffling), but if there is not enough memory allocated, RDD partitions will be evicted. To find the right amount of resources required, you should keep in mind that besides data, Spark also requires memory space for the code itself, as well as shuffle and aggregation buffers. By default, only 60% of the memory goes to RDD data, 20% to code, and the rest to buffers.

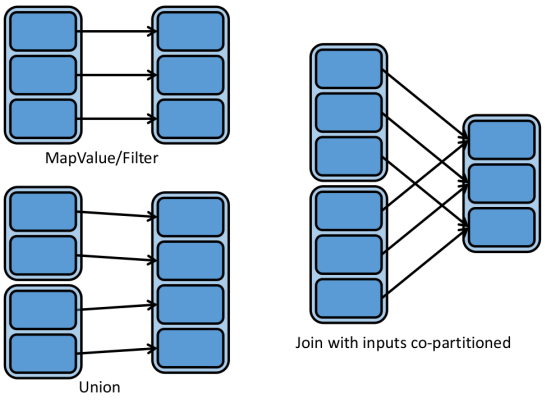


Figure 8.8: Narrow dependencies

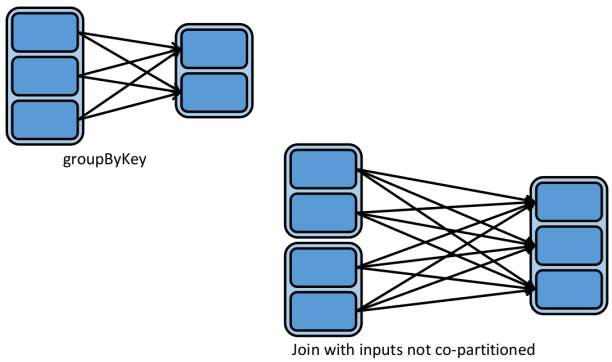


Figure 8.9: Wide dependencies

8.2.3.2 Representing RDDs

An RDD represents a distributed collection of records. These, can be of any primitive or complex data type, as long as this is serializable. As mentioned, RDD do not actually store data, but contain the necessary information required to let records flow from one operation to another and be transformed up to the DAG's sink nodes. Precisely, an RDD is composed of the following elements:

- a set of **dependencies** on parent RDD;
- the **function** that computes a new dataset based on the input from the RDD's parent;

[6] http://mesos.apache.org
[7] http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html
[8] https://kubernetes.io

- a set of **partitions**, which are the distribution units of the dataset;
  - metadata about its **partitioning scheme**, indicating what kind of partitioning strategy is being used (i.e., *hash* or *range*); and
  - the **data placement**, which denotes where each partition is being processed.

For instance, an RDD resulting from reading an HDFS file will have a partition for each block of the file, and will respect the placement with respect to the HDFS Datanode where this block has been read from. In general, some operators have a parameter to indicate the number of partitions in their output, but we can also fix the number of partitions at a given moment by explicitely calling `partitionBy` indicating the number of partitions. If you do not really want to define a new repartitioning strategy (which clearly incurs in an extra cost), `coalesce` allows to simply reduce the number of partitions by merging those in the same machine. Anyway, it is important to keep in mind the impact of parallelism in performance. On the one hand, having too few partitions hinders work balance and can even lead to resource underuse, but on the other hand, having too many partitions also can result in unnecessary and significant overheads.

In general, we can partition by either hash or range of keys, but what it specially relevant in considering using the same key and the same partitioning strategy in different RDD, because some wide dependencies (e.g., join) are this way transformed on narrow dependencies. Indeed, we can never know a priori the machine where a key will be, but since this is abolutely deterministic, given a concrete key present in two different RDD, if both use the same partitioning strategy, function and number of partitions, all the rows containing that key will end up in the same machine (irrespectively of the RDD where they belong). Thus, joining that key will not require any shuffling. It is also important to highlight that some transformation invalidate the key. For example, on the view of a `map` transformation the optimizers understands that the key may change (and consequently partitioning information is invalidated). Instead, if we use `mapValue`, we are informing the optimizer that partitioning is not affected by this transformation and consequently its annotations are still valid.

8.2.3.3 Types of RDDs

Spark provides some specializations of RDD that exploit the underlying data structure to provide specific operations or optimize their processing. A relevant example of such specialized RDD is the `PairRDD`. This is a distributed collection of key-value pairs, where the key plays a special role in aspects such as data distribution, as we will later see. PairRDDs also enable MapReduce-style operations, where the key is automatically defined as the grouping attribute. Other kinds of RDDs are those that result from specific operations, such as the `FilteredRDD`, `MappedRDD`, or `UnionRDD`, which are created from, respectively, the `filter`, `map` and `union` operators. These carry special metadata generated by the operation.

8.2.3.4 Dependencies

We distinguish two kinds of dependencies among RDD: *narrow* and *wide* dependencies. This distinction is used to determine the synchronization barriers of a Spark job. Narrow dependencies allow the flow of data to be pipelined in the same computing node, while wide dependencies might require to shuffle data to other partitions and having to wait for all inputs to finish. A narrow dependency is that where each partition of a parent RDD is used by at most one partition of the child RDD. Oppositely, wide dependencies are those where multiple child partitions may depend on one parent partition. Figures 8.8 and 8.9 depict examples of operations that involve narrow and wide dependencies. Note that a join operation requires a wide dependency unless both parents are co-partitioned with the same function.

8.2.3.5 Execution of Spark jobs

The execution of a Spark job is based on its DAG representation and RDDs dependencies. Precisely, the scheduler examines the DAG and builds a new data structure with *stages*. This data structure can be seen as the execution plan of a Spark job. Stages are actually subgraphs of the DAG defined with the objective of maximizing the number of narrow dependencies inside, while the boundaries of the stages are the wide dependencies. Figure 8.10 depicts an example of a job with three stages. Then, for each partition in a stage the scheduler launches a *task*.

Wide dependencies require shuffling data among tasks, which potentially includes moving data over the network. In this case, data are written to the disk of the source machine, similarly to the MapReduce framework, and

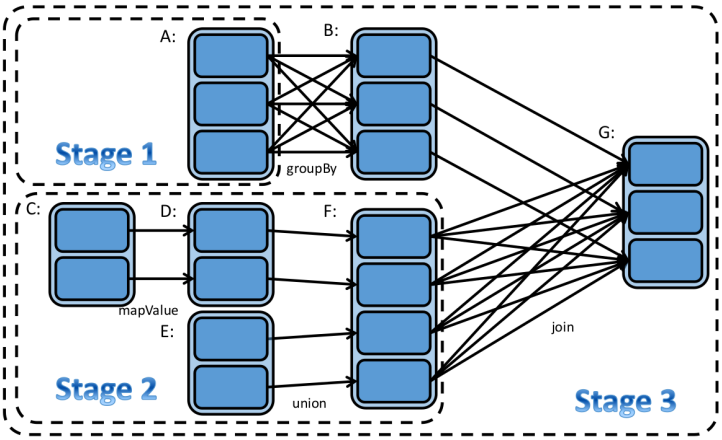


Figure 8.10: Spark execution plan with stages and tasks, from [Zah13]

it is the duty of the target machine to fetch these remotely. This also guarantees that, in the case there is a system failure, the computation can be reconstructed from such intermediate materialized results. Consequently, we can see each task as composed of three phases:

1. Fetch data (from either local or remote disk)
2. Execute operations
3. Write results (for shuffling or returning results to the driver)

8.2.3.6 Persistence

As already said, transformations are lazily evaluated, which means that they are only executed when an action is called. The effect of such on-demand strategy entails that if different actions (or even different calls to the same action) require the same transformation, this will be actually executed twice. If we want to avoid such undesirable and expensive behaviour, we can persist (a.k.a. cache) some partial results. This being really a transformation actually does not force any execution until an action is called anyway, but the second call will not require any further execution, because the persisted results will be ready to be used.

Obviously, persisting consumes memory resources that may be scarce. Therefore, there are differen configuration options depending on how much memory we want to use or if we even expect this to be swapped to the disk.`https://data-flair.training/blogs/apache-spark-rdd-persistence-caching` If we keed the data as is (i.e., deserialized), it is straightforward to use it, but requires more memory. Oppositely, if we serialize the data, these require less memory, but need to be deserialized every time they are used. Then, Spark will a LRU eviction policy in case of needing memory.

Chapter 9

Streams

A data stream is a dataset that is produced incrementally over time, instead of being fully available before its processing begins (see [GGR16]). The arrival rate of such data can be very high (e.g., tens of thousands of elements per second), and thus storing the entire stream is unfeasible. Consequently, the system must deal with data on the move using the limited memory available (see [FCKK20]) and providing the results of queries in almost real-time. Examples of use cases generating streams are Internet traffic, Share trading, Credit card use, Highway traffic monitoring, Surveilance cameras, Command and Control environments, Log monitoring, Web clicks, Sensors, and RFID.

9.1 Stream characterization

The main characteristic of streams is that their arrival rate prevents the use of iterative algorithms, since data must simply flow. The pace is definitely too fast to persist them, but in some cases it is even too fast to process every single element (some need to be ignored or skept). However, it is not only that data arrive too fast, but also that the number of elements is unbounded (potentially infinite) and their generation is not under our control (i.e., we cannot stop or delay them). Thus, whatever data structure is used to manage them, its size cannot be determined a priori and its operations need to have a very high throughput. Strictly speaking, we should not talk about real-time[1], but latency of queries should be definitely sub-second, which can only be achieved by means of parallelism and proper scalability mechanisms.

The only way to deal with all the previous difficulties is allowing approximate answers to queries. However, this does not mean that the output is not going to be deterministic. A given query (or algorithm) should always produce the same result for the same stream. It is only that such result will have a margin of error. The later is not easy, because due to geographical distribution of emitters the arrival order of elements is not necessarily the same in which they were produced. Moreover, they can be heterogeneous, contain errors or even some may be lost.

Finally, it is important to notice that streams must have some temporal locality to be of interest. This means that their content depends on time (i.e., depending on the moment of day/week/year the stream behaves differently). Indeed, data (characteristics) must evolve over time and this is what makes streaming management and analysis interesting. Otherwise, if they were always the same, one would just need to work with a static snapshot.

Stream Processing Engine	**Complex Event Processing**
Keep data moving	Pattern identification
Window aggregates definition	Pattern expressions
Handle stream imperfections	
Integrate stored and streamed data	State management
High availability of processing	High availability of patterns
Process distribution	States distribution

Table 9.1: Comparison of SPE and CEP

[1] This term should be rather reserved for safety-critical systems (e.g., self-driving cars).

9.2 Stream management

A Data Stream Management system is defined by M. Stonebraker as the "*Class of software systems that deals with processing streams of high volume messages with very low latency*".

9.2.1 Kinds of Data Stream Management Systems

At this point, it is important to distinguish two different kinds of systems (whose differences are summarized in Table 9.1):

Stream Processing Engines (SPE) focus on highly available (near) real-time processing and scalability (by means of distribution and parallelism). Their strength is to process data non-stop, mainly using relatively simple window aggregates. They also facilitate plugging the stream to other databases (e.g., for lookup). Some examples are Spark streaming, Flink, Storm, and S4.

Complex Even Processing (CEP) offer rich windowing operations to define indicators based on thresholds and express complex temporal correlations or patterns. These patterns sometimes need to be detected in long periods of time. Thus, the system needs to keep the corresponding long and complex states, which complicates their distribution and parallelization. Some examples are Esper, Aleri, StreamBase, T-Rex, Huawey PME, and Orange CRS network monitoring.

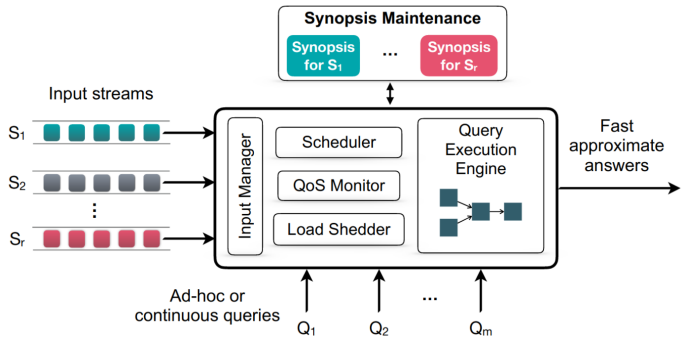


Figure 9.1: High-level architecture of an SPE, from [FCKK20]

In the following, we will focus on SPE, whose high-level architecture is sketched in Figure 9.1, and whose requirements are defined in [GÖ03] as follows:

i) Streaming query plans may not use blocking operators that must consume the entire input before any results are produced.

ii) Due to performance and storage constraints, backtracking over a data stream is not feasible. On-line stream algorithms are restricted to making only one pass over the data.

iii) The data model and query semantics must allow order-based and time-based operations (e.g., queries over a five-minute moving window).

iv) The inability to store a complete stream suggests the use of approximate summary structures (a.k.a. synopsis). As a result, queries over the summaries may not return exact answers.

v) Applications that monitor streams in real-time must react quickly to unusual data values.

vi) Long-running queries may encounter changes in system conditions throughout their execution lifetimes (e.g. variable/skewed arrival rates).

vii) Shared execution of many continuous queries is needed to ensure scalability.

	Database management	Stream management
Data	Persistent	Volatile
Access	Random	Sequential
Queries	One-time	Continuous
Support	Unlimited disk	Limited RAM
Order	Current state	Sorted
Ingestion rate	Relatively low	Extremely high
Temporal requirements	Little	Near-real time
Accuracy	Exact data	Imprecise data
Heterogeneity	Structured data	Imperfections
Algorithms	Multiple passes	One pass

Table 9.2: Comparison of database and stream management

Table 9.2 summarizes the differences between a relatively static database and managing streams. The main one is that the former persists the data, while the second does not, which somehow entails that the latter only allows sequential access (since it cannot go back and forward because elements are somehow lost). DBMS obviously persist the data in (unlimited but slow) disk, which limits the ingestion rate. Regarding retrieval, response time is not that strong requirement in DBMS as it is in SPE, but the former allow basically any kind of processing while the latter are limited to one pass over data. This is even harder, given the imprecisions and imperfections assumed in streams, which are usually not present in databases that can go though a pre-process cleaning and structuring the data.

9.2.1.1 Kinds of stream operations

Besides the problems in performance, there are also some time-based operations that are usually not considered or not possible in RDBMS, and require specific implementations. Thus, we can classify stream operations attending to three independent criteria:

Trigger: Since data is continuously flowing, we can either keep the queries continuously running or alternatively launch them at some concrete instant. Notice that the former would not make much sense in a DBMS whose data are rather static.

Outputs: The result of a query can be a set of elements (either bounded or not), but can also be a simple boolean interpreted as some kind of alert that detects a change in the behaviour of the stream.

Inputs: Queries can obviously be evaluated over a subset of data (a.k.a. window) in the stream (the subset can be a single element, too), but this is somehow equivalent to analysing small datasets. If we want to analyse the unbounded past stream as a whole, we need to rely in some summary structure (a.k.a. synopsis or sketches).

9.2.1.1.1 Window-based operations The most characteristic operation of stream processing is that of creating windows. We can simply see this as freezing or taking a snapshot of the stream. Once this is done, we can perform any operation as if working with a list of static messages. The point of doing this is getting rid of the problem of having the data on the move. However, simply taking one snapshot and never changing it would result oversimplistic. Therefore, what we take is a neverending (i.e., sliding) sequence of snapshots (like a movie of 24 photograms per second) and perform the analysis (e.g., average, min, max) one snapshot at a time. Consequently, a window is defined by both its duration and its sliding interval. If both coincide, it is called tumbling window.

9.2.1.1.2 Binary operations At this point, it should be clear that Relations and Streams are two completely different data abstractions. If we consider binary data operations (instead of unary) in an exhaustive way, we have four possible ways of joining them: Relation-Relation, Relation-Stream, Stream-Relation and Stream-Stream. Actually, the last option does not make much sense. If we consider a continuous flow of elements, and another continuous flow of elements in parallel (both potentially infinite), there are not much chances that the two elements that match are in exactly the same position at the same time. In a similar way, if we try to join two streams by just comparing the current element in both of them, these would never match.

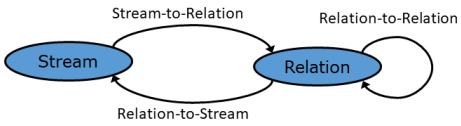


Figure 9.2: Stream binary operators

Consequently, only the three options sketched in Figure 9.2 are actually possible. Obviously, Relation-Relation operations are not that interesting at this point. Regarding the other two options, the only difference is in the order, which is not that much relevant either.

Then, if we consider Stream-Relation operations, we have to say that specific algorithms are required different from those that only deal with Relations, because one of the inputs is not static. An example is the MeshJoin algorithm (see [PSV+08]), that continuously iterates the Relation operand to optimize the cost of joining a Stream against it. Thus, if we need to join two streams, we have to take a snapshot of one of them and reduce it to the case of joining a Stream and a Relation.

9.2.1.2 Managing streams on a RDBMS

Using an RDBMS is still possible in few exceptional cases (if requirements can be relaxed). Active databases were actually the precursors of SPEs. These had the goal of automatically triggering a response to monitored events such as database updates, points in time and events external to the database. The kind of operations that such databases provide are known as ECA rules (i.e., Event-Condition-Action). These have been generally implemented in RDBMS via *triggers*, with the main objective of maintaining integrity constraints as well as derived information. However, these fall short to answer more complex aggregations over time. Furthermore, ACID transactions encompass a large overhead on data ingestion (specially to guarantee isolation and durability), which hinders the capacity of processing large data streams that arrive at a very high pace.

Nevertheless, in some cases where the arrival rate is not that high, we may use temporary tables available in many RDBMSs, whose operations are much faster than in regular tables because they are single user (i.e., they do not require isolation) and only kept in memory (not persisted in the disk, hence not guaranteeing durability). This allows to keep a tuple for every message in the stream. The table would then correspond to a sliding window over the stream, since tuples can be inserted (at arrival) and deleted (based on some aging pattern). The table can then be queried as any other, but it is only available inside a specific transaction or user session. Besides the limitation in the persistency, there could be other limitations depending on the concrete DBMS (e.g., not allowing some kind of indexing, partitioning, parallelism, or constraint checking).

9.2.1.3 Spark Streaming

Indeed, some cases resembling streams but with much weaker requirements in terms of small latency and high throughput can be solved in an RDBMS. However, if we want to scale to hundreds of nodes and guarantee sub-second latency, we need a specialized engine.

Figure 9.3 depicts how Spark (see Section 8.2) can be extended to deal with streams. Actually, Spark does not really deal with streams but with micro-batches (a.k.a. Discretized stream or DStream). Each micro-batch is just a set of messages in the stream, which can be made as small as you want (i.e., t milliseconds). Theoretically, you could create one batch per message, but obviously this would not work in practice. Nevertheless, batches can be made small enough to result very efficient and allow dealing with high arrival rate streams with very small overhead or delay.

From an implementation point of view, each micro-batch corresponds to an RDD. Thus, from some perspective, the only thing that Spark streaming does on top of Spark is splitting the stream in independent RDD. Once this is done, RDD are treated as usual and benefit from any operator or feature of generic RDD, like replication and fault tolerance. Moreover, some operators and features specific for streams are added to deal with stateful processing.

Transformations besides those already in basic Spark:

window(windowDur, slidingDur) creates a new DStream were each RDD belongs now to a window given a duration and sliding leap.

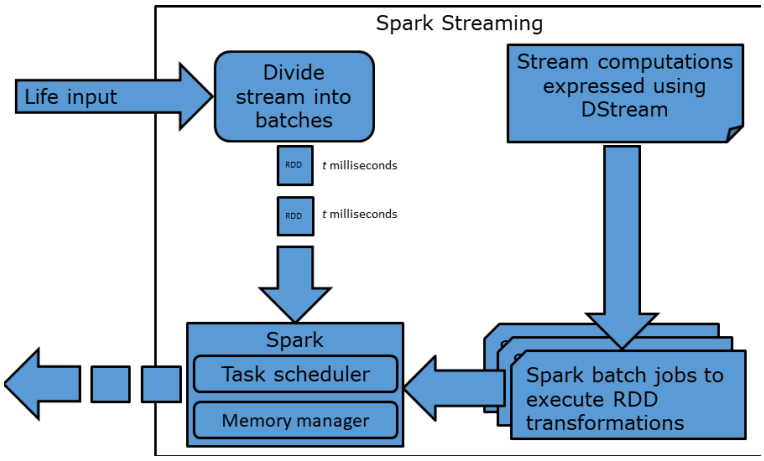


Figure 9.3: Spark streaming architecture

reduceByKeyAndWindow(aggregation, invAggregation, windowDur, slidingDur) executes the aggregation function (e.g., sum) for every key in the window. Notice that a second function (e.g., substract) is necessary to discount the elements leaving the window, so the aggregation must be invertible.

updateStateByKey(function) keeps the state of every key seen in the stream, and for every RDD goes through all of them applying the corresponding function to the value considering the arrivals in the current RDD (i.e., the function is executed once for every key in the state).

mapWithState(function) keeps the state of every key seen in the stream, and for every key-value pair in the current RDD, applies the function updating the corresponding key in the state (i.e., the function is executed once per key-value pair in the RDD).

Actions besides those already in basic Spark:

saveAsTextFiles(prefix, suffix) saves each RDD as a text file, using string representation of elements.

foreach(sparkCode) executes the Spark code for every RDD in the stream.

9.2.2 Architectural patterns

A typical approach in IT is divide-and-conquer. Indeed, difficult problems, like streams, are analyzed and solved in parts that combined together provide a solution to the overall problem. Thus, we can pay attention to different difficulties potentially posed by streams and how they could be solved isolatedly. Then, when these difficulties are found together, we just need to combine or chain the corresponding solutions.

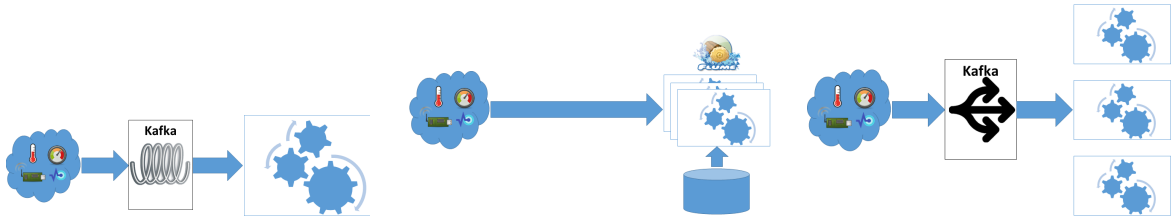


Figure 9.4: Architectural patterns

Irregular arrival is the first problem to deal with. It is not only that we get tens of thousands of messages per second in the average, but you might get hundreds of thousands per second at a given point in time. It does not make any sense to dimension your system for the worse case, and the elasticity of the Cloud is some times not agile

enough. In these cases, we need to introduce some queuing mechanisms (e.g., Kafka[2] as at LHS of Figure 9.4) to smooth the arrival rate and guarantee that no message is lost.

Getting **Near-real time** response is not always easy even for very simple processing. For example, making a database lookup to complete or complement the data in every message can compromise this requirement. A simple light-weight processing system (e.g., Flume[3] as in the middle of Figure 9.4) can help. In other cases with more complex processing, this might not be enough and the solution is a more drastic distribution approach that guarantees some locality is needed (e.g., if the same keys always go to the same machine, caching effect can improve). In these cases, we can again use different queues (e.g., Kafka as RHS of Figure 9.4) to efficiently distribute the data to multiple engine instances.

Detecting **Complex event patterns** requires specialized software (i.e., CEP) as explained above. Notice that the problem here is not only the response time, but rather to keep track of the behaviour for long enough so that the patterns are not diluted in streams with an extremely high volume of messages. Of course, this must be done keeping both the update and query time of the underlying data structure at an acceptable response rate. Dealing with the loss of some events or imperfections also complicates the problem.

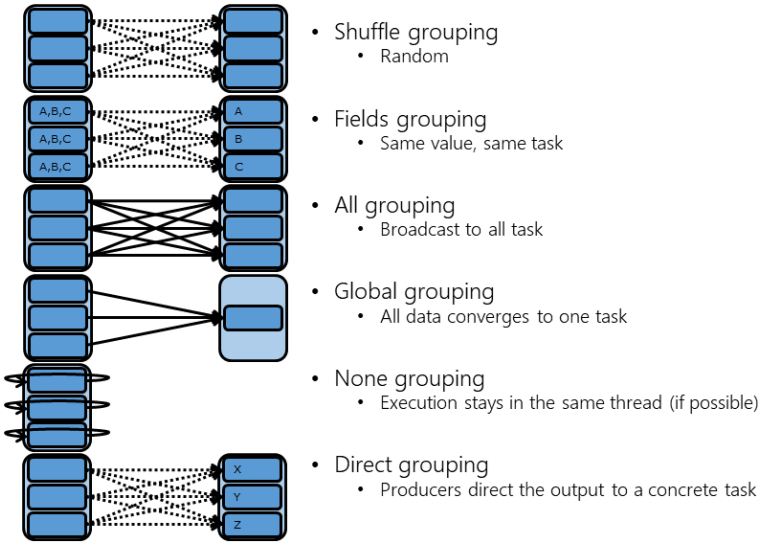


Figure 9.5: Message flow patterns

Dealing with **Heavy processing** cannot be avoided some times. In these cases, we need a system flexible enough to provide control of message routing (e.g., Storm[4]). The idea in this case is that the developer has the tools to decide where each message goes after being processed by a given machine. Thus, we can consider different patterns (sketched in Figure 9.5):

Shuffle grouping picks randomly the next machine to continue the processing of every message. This is a good option to balance the workload.

Fields grouping decides the machine where to process a message based on some of its fields. This allows to partition the load so that the messages of the same entity (e.g., credit card) are processed in the same machine, which can then easily keep track of its state (e.g., useful in fraud detection).

All grouping broadcasts every message to all machines. This can be used, for example, to try multiple processing alternatives in all messages.

[2] `https://kafka.apache.org`
[3] `https://flume.apache.org`
[4] `http://storm.apache.org`

Global grouping sends all the messages to be processed by a single machine. In this way, we can reduce the amount of resources used, when no more parallelism is needed.

None grouping retains the execution in the same machine or thread (if possible). This avoids unnecessary movement of data and also guarantees that the information used in previous processing steps is available in the next one.

Direct grouping sends the message to a concrete worker. It can use information that is not in the message (e.g., the current workload of each machine) to decide where to send it.

9.3 Stream analysis

Most of the analysis techniques are based on some summary known as *data synopses* (or *data sketches*), which are approximate compact data structures that allow for efficient lookup and updates, informing about the state of the stream from a particular perspective (e.g., number of distinct values, presence/absence of a value, etc.). If we need to cope with any arrival rate and latency of queries have to be sub-second, we must keep bounded the response time for both synopsis update and response retrieval. However, the structures are not enough and we need specific algorithms to deal with them. Due to the impossibility of storing a data stream, we have to also limit ourselves here to one-pass algorithms. Thus, the need to efficiently analyze such streams has generated a large line of research on (sub)linear algorithms to reason about the stream and its data. The main characteristics of these algorithms are that they work under (a) bounded processing time, (b) bounded resources (i.e., memory required by the corresponding data structure is, at most, logarithmic on the size of the stream), and (c) the answer must be available at any time (i.e., we do cannot wait for the stream to "finish", because it will not). Notice that some one-pass algorithms spoil the summary structure while answering the query, which would not be acceptable for streams.

Existing methods are proved to provide answers with a small probability of error, which can be pre-determined (i.e., parametrized so that it can be fine tuned by the user) given the amount of available memory and required response time. We can study the analysis techniques depending on whether they pay more attention to a limited computation or memory capacity:

a) Limited computation capacity entails that we cannot process all arriving elements. Thus, we have two options to discard them:

   i) Probabilistically drop (a.k.a. sample) some stream elements (e.g., Load shedding).

   ii) Use some characteristic of the data to filter out some elements (e.g., by means of Bloom filters, see [BM03]).

b) Limited memory capacity

   i) Focus on only part of the stream (a.k.a. Sliding window), ignoring the rest (e.g., by some aging criteria and mechanisms).

   ii) Weight the elements in the stream depending on their position (e.g., using an exponentially decaying window).

   iii) Keeping a sumary structure (a.k.a. Synopsis) that allows to approximate the response to some specific queries (e.g., Histograms, Concise sampling, Heavy hitters, Sketching).

Bibliography

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In *ICDT*, 1997.

[AG08] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley, 1995.

[AKM+16] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing access methods: The RUM conjecture. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 461–466. OpenProceedings.org, 2016.

[Amb03] S. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley& Sons, 2003.

[AMH08] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980. ACM, 2008.

[AMR+11] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2011.

[AR18] Alberto Abelló and Oscar Romero. *Online Analytical Processing*. Springer, 2018.

[BL11] Antonio Badia and Daniel Lemire. A call to arms: revisiting database design. *SIGMOD Rec.*, 40(3):61–69, 2011.

[BM03] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Math.*, 1(4):485–509, 2003.

[Car75] Alfonso F. Cardenas. Analysis and performance of inverted data base structures. *ACM Commun.*, 18(5):253–263, 1975.

[Cat10] Rick Cattell. Scalable SQL and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, 2010.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[CIF02] *Corporate Information Factory*. John Wiley&Sons, 2002.

[CK85] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, pages 268–279. ACM, 1985.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM*, 13(6):377–387, 1970.

[CZ14] C. L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Inf. Sci.*, 275:314–347, 2014.

[DG92] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150. USENIX Association, 2004.

[Dha13] Vasant Dhar. Data science and prediction. *Commun. ACM*, 56(12):64–73, 2013.

[DN14] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *VLDB J.*, 23(3):355–380, 2014.

[FCKK20] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *CoRR*, abs/2008.00842, 2020.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43. ACM, 2003.

[GGR16] Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi, editors. *Data Stream Management - Processing High-Speed Data Streams*. Data-Centric Systems and Applications. Springer, 2016.

[GMUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2009.

[GÖ03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[GRH+13] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1197–1208. ACM, 2013.

[Gun07] Neil J. Gunther. *Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services*. Springer, 2007.

[HAR16] Victor Herrero, Alberto Abelló, and Oscar Romero. NOSQL design for analytical workloads: Variability matters. In Isabelle Comyn-Wattiau, Katsumi Tanaka, Il-Yeol Song, Shuichiro Yamamoto, and Motoshi Saeki, editors, *Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings*, volume 9974 of *Lecture Notes in Computer Science*, pages 50–64, 2016.

[HAVZ20] Moditha Hewasinghage, Alberto Abelló, Jovan Varga, and Esteban Zimányi. Docdesign: Cost-based database design for document stores. In Elaheh Pourabbas, Dimitris Sacharidis, Kurt Stockinger, and Thanasis Vergoulis, editors, *SSDBM 2020: 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, July 7-9, 2020*, pages 27:1–27:4. ACM, 2020.

[HCC+13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 1223–1231, 2013. `http://www.cs.cmu.edu/%7Eqho/ssp_nips2013.pdf`.

[HNA20] Moditha Hewasinghage, Sergi Nadal, and Alberto Abelló. On the performance impact of using json, beyond impedance mismatch. In Jérôme Darmont, Boris Novikov, and Robert Wrembel, editors, *New Trends in Databases and Information Systems - ADBIS 2020 Short Papers, Lyon, France, August 25-27, 2020, Proceedings*, volume 1259 of *Communications in Computer and Information Science*, pages 73–83. Springer, 2020.

[Jar77] Donald Jardine. *The ANSI/SPARC DBMS Model*. North-Holland, 1977.

[Joh09] Ryan Johnson. Pipeline. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2116–2117. Springer US, 2009.

[JQD11] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: right shoes for a running elephant. In *SOCC*, page 21, 2011.

[Kip15] Scott Kipp. Will ssd replace hdd? `http://www.ieee802.org/3/CU4HDDSG/public/sep15/Kipp_CU4HDDsg_01a_0915.pdf`, 2015.

[KSB+99] David R. Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Comput. Networks*, 31(11-16):1203–1213, 1999.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[LBKS16] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. Lynx: a learning linux prefetching mechanism for SSD performance model. In *NVMSA*, pages 1–6, 2016.

[MAR+20] Rana Faisal Munir, Alberto Abelló, Oscar Romero, Maik Thiele, and Wolfgang Lehner. A cost-based storage format selector for materialized results in big data frameworks. *Distributed Parallel Databases*, 38(2):335–364, 2020.

[MB11] Erik Meijer and Gavin M. Bierman. A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4):49–58, 2011.

[Moh13] Chandrasekaran Mohan. History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT*, 2013.

[MRA+16] Rana Faisal Munir, Oscar Romero, Alberto Abelló, Besim Bilalli, Maik Thiele, and Wolfgang Lehner. Resilientstore: A heuristic-based data format selector for intermediate results. In *MEDI*, volume 9893 of *Lecture Notes in Computer Science*, pages 42–56. Springer, 2016.

[OCGO96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[Olt20] Dan Olteanu. The relational data borg is learning. *Proc. VLDB Endow.*, 13(12):3502–3515, 2020.

[ÖV20] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[Pit18] Evaggelia Pitoura. Pipelining. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[PRS+16] F. Pezoa, J. L. Reutter, F. Suárez, M. Ugarte, and D. Vrgoc. Foundations of JSON Schema. In *WWW*, 2016.

[PSV+08] Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils-Erik Frantzell. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. Knowl. Data Eng.*, 20(7):976–991, 2008.

[PZ11] Hasso Plattner and Alexander Zeier. *In-Memory Data Management*. Springer, 2011.

[RHAF15] Oscar Romero, Victor Herrero, Alberto Abelló, and Jaume Ferrarons. Tuning small analytics on Big Data: Data partitioning and secondary indexes in the Hadoop ecosystem. *Inf. Syst.*, 54:336–356, 2015.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 553–564. ACM, 2005.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[SS20] Stefanie Scherzinger and Sebastian Sidortschuck. An Empirical Study on the Design and Evolution of NoSQL Database Schemas. *CoRR*, abs/2003.00054, 2020.

[Sto08] Michael Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

[TADP21] Ciprian-Octavian Truic, Elena-Simona Apostol, Jrme Darmont, and Torben Bach Pedersen. The forgotten document-oriented database management systems: An overview and benchmark of native xml dodbmses in comparison with json dodbmses. *Big Data Research*, 25:100205, Jul 2021.

[Tan18] Kian-Lee Tan. *Distributed Database Systems*, pages 894–896. Springer US, 2018.

[TCGM17] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. Enabling query processing across heterogeneous data models: A survey. In Jian-Yun Nie, Zoran Obradovic, Toyotaro Suzumura, Rumi Ghosh, Raghunath Nambiar, Chonggang Wang, Hui Zang, Ricardo Baeza-Yates, Xiaohua Hu, Jeremy Kepner, Alfredo Cuzzocrea, Jian Tang, and Masashi Toyoda, editors, *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 3211–3220. IEEE Computer Society, 2017.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[W3C04] W3C. *Extensible Markup Language (XML) 1.0*, 3rd edition, February 2004.

[Zah13] Matei A. Zaharia. *An Architecture for and Fast and General Data Processing on Large Clusters*. PhD thesis, University of California, Berkeley, USA, 2013.

Appendix A

Acronyms

ACID Atomicity, Consistency, Isolation and Durability
BASE Basically Available, Soft state, Eventual consistency
BSP Bulk Synchronous Parallel model
CSV Comma-separated Values
DAG Directed Acyclic Graph
DB Database
DBA Database Administrator
DBMS Database Management System
DE Data Engineering
DFS Distributed File System
DS Data Science
DW Data Warehousing
FK Foreign Key
FS File System
HTAP Hybrid Transaction/Analytical Processing
I/O Input/Output
JVM Java Virtual Machine
KV Key/Value
LAN Local Area Network
LHS Left Hand Side
LRU Least Recently Used
NOSQL Not Only SQL
OLAP On-Line Analytical Processing
OLTP On-Line Transactional Processing
OS Operating System
PK Primary Key
RDBMS Relational DBMS
RDD Resilient Distributed Dataset
RHS Right Hand Side
SQL Structured Query Language
WAN Wide Area Network