Data Warehousing and OLAP

Alberto Abelló Petar Jovanovic

Database Technologies and Information Management (DTIM) group
Universitat Politècnica de Catalunya (BarcelonaTech), Barcelona
August 22, 2023

Foreword

Information assets are immensely valuable to any enterprise, and because of this, these assets must be properly stored and readily accessible when they are needed. However, the availability of too much data makes the extraction of the most important information difficult, if not impossible. View results from any Google search, and you'll see that the "data = information" equation is not always correct. That is, too much data is simply too much.

Data warehousing is a phenomenon that grew from the huge amount of electronic data stored (starting in the 90s) and from the urgent need to use those data to accomplish goals that go beyond the routine tasks linked to daily processing. In a typical scenario, a large corporation has many branches, and senior managers need to quantify and evaluate how each branch contributes to the global business performance. The corporate database stores detailed data on the tasks performed by branches. To meet the managers' needs, tailor-made queries can be issued to retrieve the required data. In order for this process to work, database administrators must first formulate the desired query (typically an SQL query with aggregates) after closely studying database catalogs. Then the query is processed. This can take a few hours because of the huge amount of data, the query complexity, and the concurrent effects of other regular workload queries on data. Finally, a report is generated and passed to senior managers in the form of a spreadsheet.

Many years ago, database designers realized that such an approach is hardly feasible, because it is very demanding in terms of time and resources, and it does not always achieve the desired results. Moreover, a mix of analytical queries and transactional routine queries inevitably slows down the system, and this does not meet the needs of users of either type of query. Today's advanced data warehousing processes separate online analytical processing (OLAP) from online transactional processing (OLTP) by creating a new information repository that integrates basic data from various sources, properly arranges data formats, and then makes data available for analysis and evaluation aimed at planning and decision-making processes.

For example, some fields of application for which data warehouse technologies are successfully used are:

- Trade: Sales and claims analyses, shipment and inventory control, customer care and public relations
- Craftsmanship: Production cost control, supplier and order support
- Financial services: Risk analysis and credit cards, fraud detection
- Transport industry: Vehicle management
- Telecommunication services: Call flow analysis and customer profile analysis
- Health care service: Patient admission and discharge analysis and bookkeeping in accounts departments

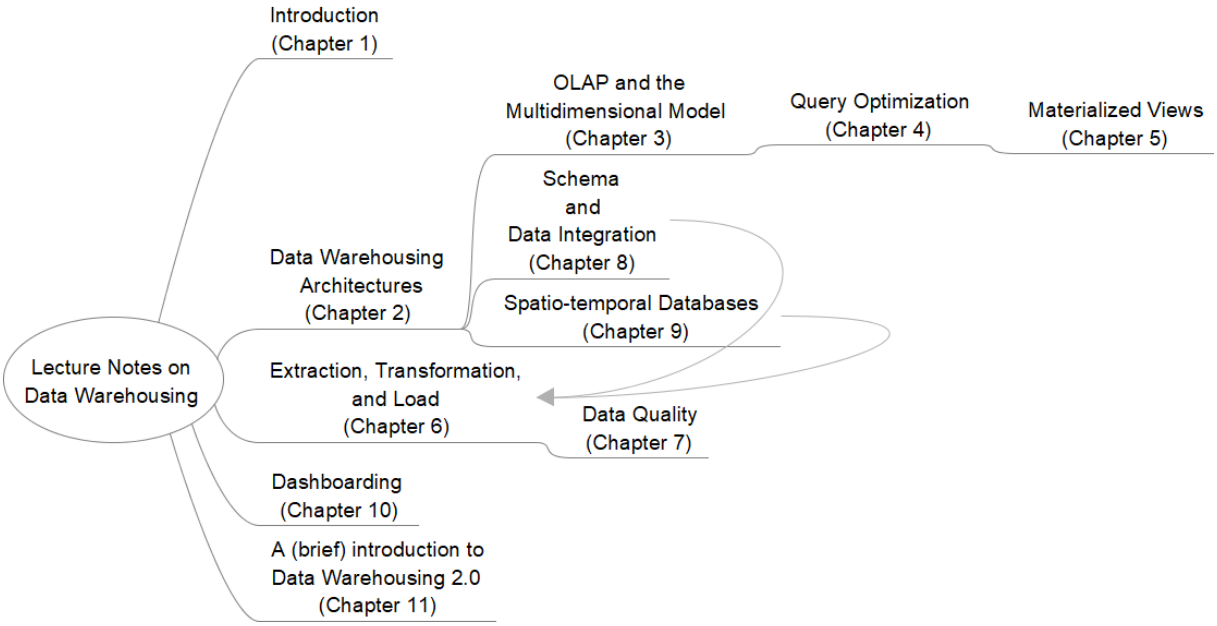
The field of application of data warehouse systems is not only restricted to enterprises, but it also ranges from epidemiology to demography, from natural science to education. A property that is common to all fields is the need for storage and query tools to retrieve information summaries easily and quickly from the huge amount of data stored in databases or made available on the Internet. This kind

of information allows us to study business phenomena, learn about meaningful correlations, and gain useful knowledge.

In the following, we present an introduction to such systems and some insights to the techniques, technologies and methods underneath.

Structure

The diagram below indicates dependencies among chapters from a conceptual viewpoint.



After the **Introduction** (Chapter 1), we can see **Data Warehousing Architectures** (Chapter 2). Then, following the four characteristics of a DW defined by W. Inmon (namely subject orientation, integration, historicity and non-volatility), we can find chapters about **OLAP and the Multidimensional Model** (Chapter 3, enabling subject-oriented data analysis), **Schema and Data Integration** (Chapter 8), and **Spatio-Temporal Databases** (Chapter 9, covering bi-temporal databases implementing historicity and non-volatility). Related to relational implementations of OLAP tools, we have **Query Optimization** (Chapter 4), and as a specific optimization technique **Materialized Views** (Chapter 5). As a next phase in DW, we have the building of the **Extraction, Transformation and Load** flows (Chapter 6), which would involve dealing with **Data Quality** issues (Chapter 7), as well as the implementation of the integration and bitemporality mentioned before. The last phase, making use of the data, involves the visualization in the form of **Dashboarding** (Chapter 10). Finally, there is a view of improvements or extensions in the form of **A (brief) introduction to Data Warehousing 2.0** (Chapter 11).

Contents

Chapter 1

Introduction

Nowadays, the *free market economy* is the basis of *capitalism* (the current global economic system) in which the production and distribution of goods are decided by market businesses and consumers; giving rise to the *supply and demand* concept. In this scenario, being more competitive than the other organizations becomes essential, and *decision making* raises as a key factor for the organization success.

Decision making is based on *information*. The more accurate information I get, the better decisions I can make to get competitive advantages. That is the main reason why information (understood as the result of processing, manipulating and organizing data in a way that adds new knowledge to the person or organization receiving it) has become a key piece in any organization. In the past, managers' ability for foreseeing upcoming trends was crucial, but this largely subjective scenario changed when the world *became digital*. Actually, any event can be recorded and stored for later analysis, which provides new and objective business perspectives to help managers in the decision making process. Hence, (digital) information is a valuable asset to organizations, and it has given rise to many well-known concepts such as *Information Society*, *Information Technologies* and *Information Systems* among others.

For this reason, today, decision making is a research hot topic. In the literature, those applications and technologies for gathering, providing access to, and analyzing data for the purpose of helping organization managers make better business decisions are globally known as *Decision Support Systems*, and those computer-based techniques and methods used in these systems to extract, manipulate and analyze data as *Business Intelligence* (BI). Specifically, BI can be defined as a broad category of applications and technologies for gathering, integrating, analyzing, and providing access to data to help enterprise users make better business decisions. BI applications include the activities of decision support systems, query and reporting, online analytical processing (OLAP), statistical analysis, forecasting, and data mining.

BI implies having a comprehensive knowledge of any of the factors that affect an organization business with one main objective: the better decisions you make, the more competitive you are. Under the BI concept we embrace many different disciplines such as *Marketing*, *Geographic Information Systems* (GIS), *Knowledge Discovery* or *Data Warehousing*.

In this work, we focus on the latter, which is, possibly, the most popular, generic solution to give answer to extract, store (conciliate) and analyze data (what is also know as the BI cycle). Fig. 1.1 depicts a data warehousing system supporting decision-making. There, data extracted from several data sources is first transformed (i.e., cleaned and homogenized) prior to be loaded in the data warehouse. The data warehouse is a read-only database (meaning that data loading is not performed by the end-users, who only query them), intended to be exploited by end-users by means of the exploitations tools (such as query and reporting, data mining and On-Line Analytical Processing -OLAP-). The analysis carried out over the data warehouse represents valuable, objective input used in the organizations to build their business strategy. Eventually, these decisions will impact on the data sources collecting data about organizations and, again, we repeat the cycle to analyze our business reality and come up with an adequate strategy suiting our necessities.

In the following sections, we discuss in detail data warehousing systems and we will also focus on their main components, such as the data warehouse, the metadata repository, the ETL tools and the most relevant exploitation tool related to these systems: OLAP tools.

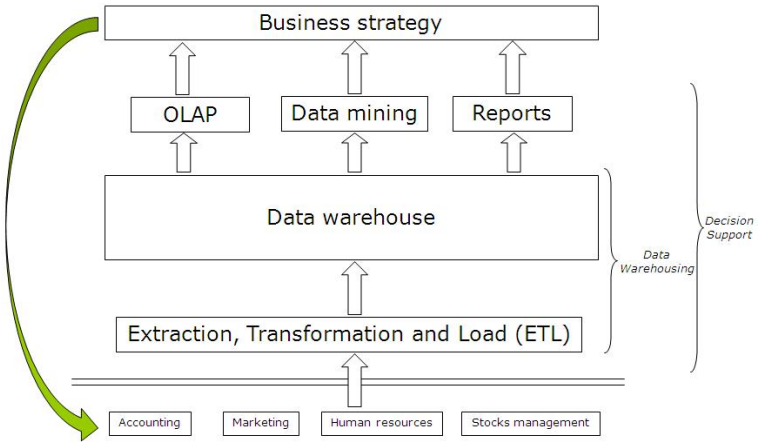


Figure 1.1: Business Intelligence cycle

1.1 Data Warehousing Systems

Data warehousing systems are aimed at exploiting the organization data, previously integrated in a huge repository of data (the *data warehouse*), to extract relevant knowledge of the organization.

A formal definition would be: *Data Warehousing is a collection of methods, techniques and tools used to support* knowledge workers *-senior managers, directors, managers and analysts- to conduct data analyses that help with performing decision-making processes and improving information resources.* This definition gives a clear idea of these systems final aim: give support to decision making without regard of technical questions like data heterogeneity or data sources implementation. This is a key factor in data warehousing. Nowadays, any event can be recorded within organizations. However, the way each event is stored differs, and it depends on several factors such as relevant attributes for the organization (i.e., their daily needs), technology used (i.e., implementation), analysis task performed (i.e., data relevant for decision making), etc. Thus, these systems must gather and assemble all (relevant) business data available from various (and possibly heterogeneous) sources in order to gain a single and detailed view of the organization that later will be properly managed and exploited to give support to decision making. The role of data warehousing can be better understood with five claims introduced by Kimball in [KRTR98]:

- *We have heaps of data, but we cannot access it.* Loads of data are available. However, we need the appropriate tools to effectively exploit (in the sense of query and analyze) it.
- *How can people playing the same role achieve substantially different results?* Any organization may have several databases available (devoted to specific business areas) but they are not conceptually integrated. Providing a single and detailed view of the business process is a must.
- *We want to select, group and manipulate data in every possible way.* This claim underlines the relevance of providing powerful and flexible analysis methods.
- *Show me just what matters.* Too much information may be, indeed, too much. The end-user must be able to focus on relevant information for his / her current decision making processes.
- *Everyone knows that some data is wrong.* A sensitive amount of transactional data is not correct and it has to be properly cleaned (transformed, erased, filtered, etc.) in order to avoid misleading results.

Data warehousing systems have three main components: the data warehouse, the ETL *(Extraction, Transformation and Load*) tools and the exploitation tools. The data warehouse is a huge repository of data; i.e., a database. It is the data warehousing system core and that is why these systems are also called *data warehouse systems*. However, it is not just another traditional database: it depicts a single and detailed view of the organization business. By means of the ETL tools data from a variety of sources

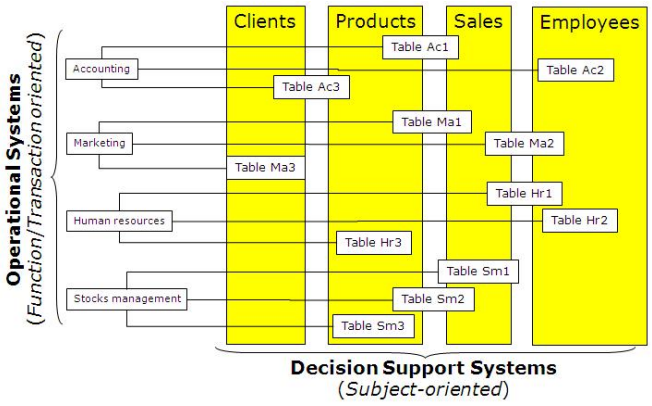


Figure 1.2: Subject oriented Vs. transaction oriented

is loaded (i.e., homogenized, cleaned and filtered) into the data warehouse. Once loaded, it is ready to be exploited by means of the exploitation tools.

1.1.1 The Data Warehouse

The data warehouse term was coined by B. Inmon in 1992 in [Inm92], which defined it as: *"a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decision making process"*. Where:

- *Subject oriented* means that data stored gives information about a particular subject instead of the daily operations of an organization. These data are clustered together in order to undertake different analysis processes over it. This fact is represented in Fig. 1.2. There, tables from the operational sources (which were thought to boost transaction performance) are broken into small pieces of data before loading the data warehouse, which is interested in concepts such as clients or products whose data can be spread all over different transactional tables. Now, we can analyze concepts such as clients that were spread on different transactional system;
- *Integrated* means that data have been gathered into the data warehouse from a variety of sources and merged into a coherent whole;
- *Time-variant* means that all data in the data warehouse is identified with a particular time period (usually called *Valid Time*, representing when data are valid in the real world) and for example, historical data (which is of no interest for OLTP systems) is essential; and finally,
- *Non-volatile* means that data are stable in the data warehouse. Thus, more data are added but data are never removed (usually a *transaction time* attribute is added to represent when things are recorded). This enables management to gain a consistent picture of the business.

Despite this definition was introduced more than 30 years ago, it still remains reasonably accurate. However, a single-subject data warehouse is currently referred to as a *data mart* (i.e., a local or departmental data warehouse), while data warehouses are more global, giving a general enterprise view. In the literature, we can find other definitions like the one presented in [KRTR98], where a data warehouse is defined as *"a copy of transaction data specifically structured for query and analysis"*; this definition, despite being simpler, is not less compelling, since it underlines the relevance of querying in a data warehousing system. The data warehouse design is focused on improving queries performance instead of improving update statements (i.e., insert, update and delete) like transactional databases do. Moreover, the data warehousing system end-users are high-ranked people involved in decision making rather than those low/medium-ranked people maintaining and developing the organization information systems. Next table summarizes main differences between an operational database and a data warehouse (or, in general, a decisional system):

	Operational	Decisional
Objective	Business operation	Business analysis
Main functions	Daily oper. (OLTP)	Decision Support System (OLAP)
Usage	Repetitive (predefined)	Innovative (unexpected)
Design orientation	Functionality	Subject
Kind of users	Clerks	Executives
Number of users	Thousands	Hundreds
Accessed tuples	Hundreds	Thousands
Data sources	Isolated	Integrated
Granularity	Atomic	Summarized
Time coverage	Current	Historical
Access	Read/Write	Read-only
Work units	Simple transactions	Complex queries
Requirements	Performance & consistency	Performance & precision
Size	Mega/Gigabytes	Giga/Tera/Petabytes

Figure 1.3: Operational Vs. decisional systems

Most issues pointed out in table 1.3 have been already discussed, but some others may still remain unclear. Operational (mainly transactional -OLTP- systems) focus on giving solution to the daily business necessities, whereas decisional systems, such as data warehousing, focus on providing a single, detailed view of the organization to help us on decision making. Usually, a decision is not made on the isolated view of current data, but putting this in the context of historical evolution, and data are consequently never updated or deleted, but only added. This addition of data is done in bulk loads performed by batch processes. Users never do it manually, so the data warehouse is considered "read-only" from this perspective. Thus, data warehouses tend to be massive (incrementally integrate historical information from different sources, which is never deleted), and nowadays we can find the first petabyte data warehouses (i.e., at least, one order of magnitude larger than OLTP systems). The reason is that data warehouses not only load data (including historical data) integrated from several sources but also pre-computed data aggregations derived from it. In decision making, aggregated (i.e., summarized) queries are common (e.g., revenue obtained per year, or average number of pieces provided per month, etc.) since they provide new interesting perspectives. This is better explained in the following section.

1.1.1.1 Kinds of Data

Data extracted from the data sources (owned databases, shared with other partners, external data coming from the Web, etc.) can be classified in many ways, but we emphasize the following three categorizations, which helps to understand the operational - decisional systems duality:

- **Operational Vs. decisional data:** Our data sources contain heaps of data. An operational system stores any detail related to our daily processes. However, not all data are relevant for decision making. For example, many retailers are interested in our postal code when shopping but they show no interest in our address, since their decision making processes deal with city regions (represented by the postal code). Another clear example is the name and surname, which are not very interesting in the general case. Oppositely, the age and gender tend to be considered as first-class citizens in decision making. Indeed, one of the main challenges to populate a data warehouse is to decide which data subset from our operational sources is needed for our decision making processes. Some data are clearly useless for decision making (e.g., customer names), some are clearly useful (e.g., postal codes), but others are arguable or not that clear (e.g., telephone numbers per se can be useless to make decisions, but result useful as identifiers of customers). Thus, the decisional data can be seen as a strategic window over the whole operational data. Furthermore, operational data quality is poor, as update transactions happen constantly and data consistency can be compromised by them. For this reason, the cleaning stage (see section 1.1.2) when loading

data in the data warehouse is crucial to guarantee reliable and exact data.

- **Historical Vs. current data:** Operational sources typically store *daily* data; i.e., current data produced and handled by the organization processes. Old data (from now on, historical data), however, was typically moved away from the operational sources and stored in secondary storage systems such as tapes in form of backups. What is current or old data depends, in the end, on each organization and its processes. For example, consider an operational source keeping track of each sale in a supermarket. For performance reasons, the database only stores a one year window. In the past, older data was dumped into backup tapes and left there, but nowadays, it is kept in the decisional systems. Note some features about this dichotomy. A decisional system obviously needs both. Posing queries regarding large time spans is common in decision making (such as what was the global revenue in the last 5 years). Both data, though, come from the same sources and current data will, eventually, become old. Thus, it is critical to load data periodically so that the decisional system can keep track of them.
- **Atomic, derived and aggregated data:** These concepts are related to the *data granularity* at which data are delivered. We refer to atomic data to the granularity stored by the operational sources. For example, I can store my sales as follows: the user who bought it, the shop where it was sold, the exact time (up to milliseconds), the price paid and the discount applied. However, decisional systems often allow to compute derived and aggregated data from atomic data to give answer to interesting business questions. On the one hand, derived data results from computing a certain function over atomic data to produce non-evident knowledge. For example, we may be interested in computing the revenue obtained from a user, which can be obtained by applying the discount over the initial price and summing up all his sales. Another example would be data produced by data mining algorithms (e.g., the customer profile defined by a clustering algorithm). Normally, different attributes or values are needed to compute derived data. On the other hand, aggregates results from applying an aggregation function for a certain value. For example, the average item price paid per user, or the global sales in 2011. Thus, it can be seen as a specific kind of derived data. Derived and aggregated data frequently queried are often pre-computed in decisional systems (although they can also be computed on-the-fly). The reason is that previous experiences suggest to pre-compute the most frequent aggregates and derived data in order to boost performance (the trade-off between update and query frequencies needs to be considered).

1.1.2 ETL Tools: Extraction Transformation and Load

The ETL processes extract, integrate, and clean data from operational sources to feed the data warehouse. For this reason, the ETL process operations as a whole are often referred as reconciliation. These are also the most complex and technically challenging among all the data warehouse process phases (ETL processes are responsible to disregard data heterogeneities of any kind). ETL takes place once when a data warehouse is populated for the first time, then it occurs every time the data warehouse is regularly updated. Fig. 1.4 shows that ETL consists of four separate phases: extraction (or capture), cleansing (or cleaning or scrubbing), transformation, and loading. In the following sections, we offer brief descriptions of these phases.

The scientific literature shows that the boundaries between cleansing and transforming are often blurred from the terminological viewpoint. For this reason, a specific operation is not always clearly assigned to one of these phases. This is obviously a formal problem, but not a substantial one. Here, cleansing is essentially aimed at rectifying data values, and transformation more specifically manages data formats.

1.1.2.1 Extraction

Relevant data are obtained from sources in the extraction phase. You can use static extraction when a data warehouse needs populating for the first time. Conceptually speaking, this looks like a snapshot of operational data. Incremental extraction, used to update data warehouses regularly, seizes the changes applied to source data since the latest extraction. Incremental extraction is often based on the log maintained by the operational DBMS. If a timestamp is associated with operational data to record exactly

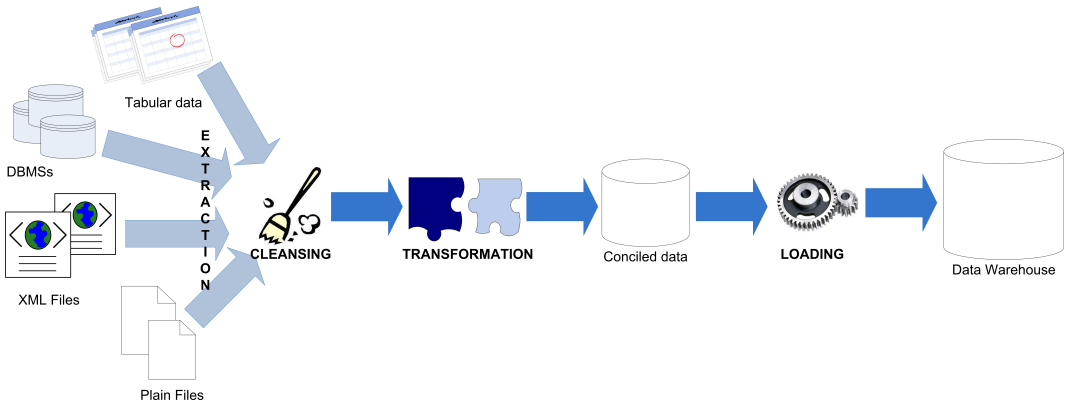


Figure 1.4: Extraction, transformation and loading

when the data are changed or added, it can be used to streamline the extraction process. Extraction can also be source-driven if you can rewrite operational applications to asynchronously notify of the changes being applied, or if your operational database can implement triggers associated with change transactions for relevant data. The data to be extracted is mainly selected on the basis of its quality. In particular, this depends on how comprehensive and accurate the constraints implemented in sources are, how suitable the data formats are, and how clear the schemas are.

1.1.2.2 Cleansing

The cleansing phase is crucial in a data warehouse system because it is supposed to improve data quality, normally quite poor in sources. The following list includes the most frequent mistakes and inconsistencies that make data "dirty":

- Duplicate data: For example, a patient is recorded many times in a hospital patient management system
- Inconsistent values that are logically associated: Such as addresses and ZIP codes
- Missing data: Such as a customer's job
- Unexpected use of fields: For example, a SSN (social Security Number) field could be used improperly to store office phone numbers
- Impossible or wrong values: Such as 30/2/2009
- Inconsistent values for a single entity because different practices were used: For example, to specify a country, you can use an international country abbreviation (I) or a full country name (Italy); similar problems arise with addresses (Hamlet Rd. and Hamlet Road)
- Inconsistent values for one individual entity because of typing mistakes: Such as Hamet Road instead of Hamlet Road

In particular, note that the last two types of mistakes are very frequent when you are managing multiple sources and are entering data manually. The main data cleansing features found in ETL tools are rectification and homogenization. They use specific dictionaries to rectify typing mistakes and to recognize synonyms, as well as rule-based cleansing to enforce domain-specific rules and define appropriate associations between values.

1.1.2.3 Transformation

Transformation is the core of the reconciliation phase. It converts data from its operational source format into a specific data warehouse format. Establishing a mapping between the data sources and the data warehouse is generally made difficult by the presence of many different, heterogeneous sources. If this is the case, a complex integration phase is required when designing your data warehouse. The following points must be rectified in this phase:

- Loose texts may hide valuable information. For example, BigDeal LtD does not explicitly show that this is a Limited Partnership company.
- Different formats can be used for individual data. For example, a date can be saved as a string or as three integers.

Following are the main transformation processes at this stage:

- Conversion and normalization that operate on both storage formats and units of measure to make data uniform.
- Matching that associates equivalent fields in different sources.
- Selection that reduces the number of source fields and records.

When populating a data warehouse, you most surely may need to sum up data properly and pre-compute interesting data aggregations. So that, pre-aggregated data may be needed to be computed at this point.

1.1.2.4 Loading

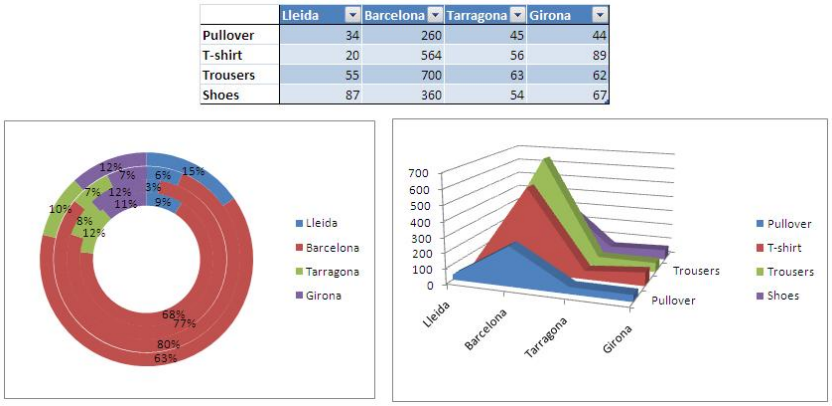
Loading into a data warehouse is the last step to take. Loading can be carried out in two ways:

- Refresh: Data warehouse data are completely rewritten. This means that older data are replaced. Refresh is normally used in combination with static extraction to initially populate a data warehouse.
- Update: Only those changes applied to source data are added to the data warehouse. Update is typically carried out without deleting or modifying preexisting data. This technique is used in combination with incremental extraction to update data warehouses regularly.

1.1.3 Exploitation Tools

The final aim of every data warehousing system is to exploit the data warehouse. The data warehouse is a huge repository of data that does not tell much by itself; like in the operational databases field, we need auxiliary tools to query and analyze data stored. In this field, those tools aimed at extracting relevant information from the repository of data are known as the exploitation tools. Without the appropriate exploitation tools, we will not be able to extract valuable knowledge of the organization from the data warehouse, and the whole system will fail in its aim of providing information for giving support to decision making. Most used exploitation tools can be classified in three categories:

- Query & Reporting: This category embraces the evolution and optimization of the traditional query & reporting techniques. This concept refers to an exploitation technique consisting of querying data and generating detailed pre-defined reports to be interpreted by the end-user. Mainly, this approach is oriented to those users who need to have regular access to the information in an almost static way. A report is defined by a query and a layout. A query generally implies a restriction and an aggregation of multidimensional data. For example, you can look for the monthly receipts during the last quarter for every product category. A layout can look like a table or a chart (diagrams, histograms, pies, and so on). Fig. 1.5 shows a few examples of layouts for a query.



	Lleida	Barcelona	Tarragona	Girona
Pullover	34	260	45	44
T-shirt	20	564	56	89
Trousers	55	700	63	62
Shoes	87	360	54	67

Figure 1.5: Query & reporting

- Data Mining: Data mining is the exploration and analysis of large quantities of data in order to discover meaningful patterns and rules. The data mining field is a research area per se, but as the reader may note, this kind of techniques and tools suit perfectly to the final goal of data warehousing systems. Typically, data from the data warehouse is dumped into plain files that feed the data mining algorithms. Thus, both techniques are traditionally used sequentially: populate the data warehouse and then, select relevant data to be analyzed by data mining algorithms. Recently, though, new techniques to tight the relationship between both worlds have been addressed, such as performing the data mining algorithms *inside* the data warehouse and avoiding the bottleneck produced by moving data out. However, this kind of approaches stay out of this course contents.
- OLAP Tools: OLAP stands for *On-Line Analytical Processing*, which was carefully chosen to confront the OLTP acronym (*On-Line Transactional Processing*). Its main objective is to analyze business data from its dimensional or components perspective; unlike traditional operational systems such as OLTP systems. For a deep insight on OLAP, see chapter 3.

Multimedia Materials

Data Warehouse definition (en)

Data Warehouse definition (es)

Chapter 2

Data Warehousing Architectures

The following architecture properties are essential for a data warehouse system[1]:

- **Separation:** Analytical and transactional processing should be kept apart as much as possible.
- **Scalability:** Hardware and software architectures should be easy to upgrade as the data volume, which has to be managed and processed, and the number of users' requirements, which have to be met, progressively increase.
- **Extensibility:** The architecture should be able to host new applications and technologies without redesigning the whole system.
- **Security:** Monitoring accesses is essential because of the strategic data stored in data warehouses.
- **Administerability:** Data warehouse management should not be overly difficult.

In the following, sections 2.1, 2.2, and 2.3 present a structure-oriented classification that depends on the number of layers used by the architecture.

2.1 Single-layer Architecture

A single-layer architecture is not frequently used in practice. Its goal is to minimize the amount of data stored; to reach this goal, it removes data redundancies. Fig. 2.1 shows the only layer physically available: the source layer. In this case, data warehouses are virtual. This means that a data warehouse is implemented as a multidimensional view of operational data created by specific middleware, or an intermediate processing layer.

The weakness of this architecture lies in its failure to meet the requirement for separation between analytical and transactional processing. Analysis queries are submitted to operational data after the middleware interprets them. It this way, the queries affect regular transactional workloads. In addition, although this architecture can meet the requirement for integration and correctness of data, it cannot log more data than sources do. Thus, a relevant consideration about this approach is the nature of the sources, because depending on it, the architecture becomes impractical or even impossible (this also impacts to some extent the other architectures).

- Non computerized sources (i.e., manual extraction) is more and more rare, but if required, makes the single-layer architecture impossible, because implies to repeat the manual effort once and again, since extracted information is not kept for further use.
- World Wide Web, which is not under our control and frequently changes, also brings problems to the single-layer architecture, because information that disappears from the Internet will also be lost for use, if we didn't make any copy.

[1] This chapter is mainly based on the book "Data Warehouse Design: Modern Principles and Methodologies", published by McGraw Hill and authored by Matteo Golfarelli and Stefano Rizzi, 2009 [GR09].

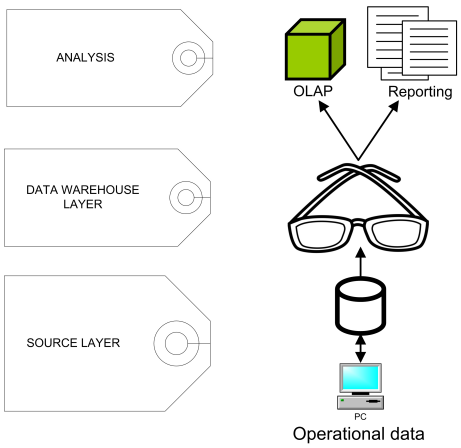


Figure 2.1: Single-layer architecture for a data warehouse system

- Partners' systems can also be available under some kind of agreement, which can include what happens in case of changes in the availability or the format of data.
- Own operational systems always result the easiest data source, because they are well known and we have control over their changes.

For these reasons, a virtual approach to data warehouses can be successful only if analysis needs are particularly restricted and the data volume to analyze is not huge.

2.2 Two-layer Architecture

The requirement for separation plays a fundamental role in defining the typical architecture for a data warehouse system, as shown in Fig. 2.2. Although it is typically called a two-layer architecture to highlight a separation between physically available sources and data warehouses, it actually consists of four subsequent data flow stages:

- **Source layer:** A data warehouse system uses heterogeneous sources of data. That means that data feeding the data warehouse might come from inside the organization or from external sources (such as the Cloud, the Web, or from partners). Furthermore, the technologies and formats used to store these data may also be heterogeneous (legacy[2] systems, Relational databases, XML files, plain text files, e-mails, pdf files, Excel and tabular tables, OCR files, etc.). Furthermore, some areas have their specificities; for example, in medicine other data inputs such as images or medical tests must be treated as first-class citizens.
- **Data staging:** The data stored to sources should be extracted, cleansed to remove inconsistencies and fill gaps, and integrated to merge heterogeneous sources into one common schema. The so-called Extraction, Transformation, and Loading tools (ETL), can merge heterogeneous schemas, extract, transform, cleanse, validate, filter, and load source data into a data warehouse. Technologically speaking, this stage deals with problems that are typical for distributed information

[2]The term legacy system denotes corporate applications, typically running on mainframes or minicomputers, that are currently used for operational tasks but do not meet modern architectural principles and current standards. For this reason, accessing legacy systems and integrating them with more recent applications is a complex task. All applications that use a pre-Relational database are examples of legacy systems.

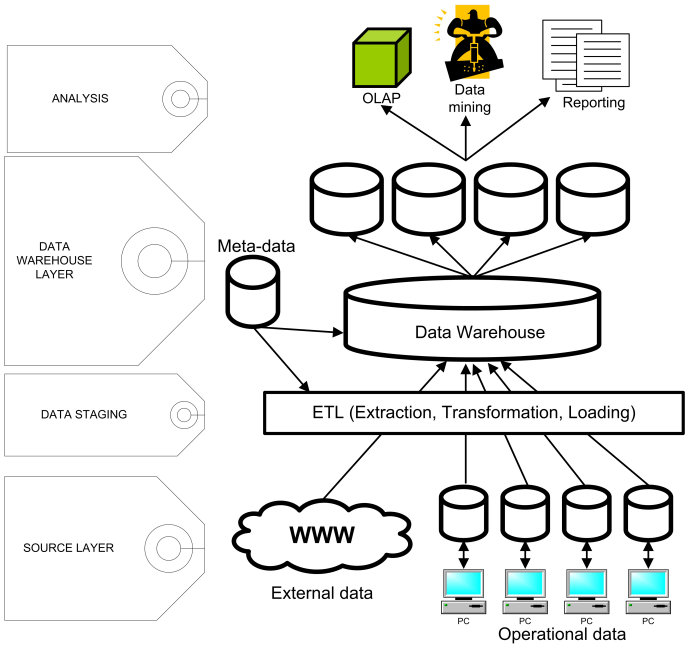


Figure 2.2: Two-layer architecture for a data warehouse system

systems, such as inconsistent data management and incompatible data structures. Section 1.1.2 deals with a few points that are relevant to data staging.

- **Data warehouse layer**: Information is stored to one logically centralized single repository: a data warehouse. The data warehouse can be directly accessed, but it can also be used as a source for creating *data marts* (in short, local/smaller data warehouses), which partially replicate data warehouse contents and are designed for specific enterprise departments. Metadata repositories (see section 2.4) store information on sources, access methods available, data staging, users, data mart schemas, and so on.

- **Analysis:** In this layer, integrated data are efficiently and flexibly accessed to issue reports, dynamically analyze information, and simulate hypothetical business scenarios. Technologically speaking, it should feature aggregate data navigators, complex query optimizers, and user-friendly GUIs. Section 1.1.3 deals with different types of decision-making support analyses.

The architectural difference between data warehouses and data marts needs to be studied closer. The component marked as a data warehouse in Fig. 2.2 is also often called the primary data warehouse or corporate data warehouse. It acts as a centralized storage system for all the data being stored together. Data marts can be viewed as small, local data warehouses replicating (and pre-computing as much as possible) the part of a primary data warehouse required for a specific application domain. More formally, A *data mart* is a subset or an aggregation of the data stored to a primary data warehouse. It includes a set of information pieces relevant to a specific business area, corporate department, or category of users. The data marts populated from a primary data warehouse are often called dependent. Although data marts are not strictly necessary, they are very useful for data warehouse systems in midsize to large enterprises because:

- they are used as building blocks while incrementally developing data warehouses;

- they mark out the information required by a specific group of users to solve queries;

- they can deliver better performance because they are smaller (i.e., only partial history, not all sources and not necessarily the most detailed data) than primary data warehouses.

Sometimes, mainly for organization and policy purposes, you should use a different architecture in which sources are used to directly populate data marts. These data marts are called independent. If there is no primary data warehouse, this streamlines the design process, but it leads to the risk of inconsistencies between data marts. To avoid these problems, you can create a primary data warehouse and still have independent data marts. In comparison with the standard two-layer architecture of Fig. 2.2, the roles of data marts and data warehouses are actually inverted. In this case, the data warehouse is populated from its data marts, and it can be directly queried to make access patterns as easy as possible. The following list sums up all the benefits of a two-layer architecture (as compared against the single-layer), in which a data warehouse separates sources from analysis applications:

- In data warehouse systems, good quality information is always available, even when access to sources is denied temporarily for technical or organizational reasons.
- Data warehouse analysis queries do not affect the management of transactions, the reliability of which is vital for enterprises to work properly at an operational level.
- Data warehouses are logically structured according to the multidimensional model (see chapter 3), while operational sources are generally based on Relational or semi-structured models.
- A mismatch in terms of time and granularity occurs between OLTP systems, which manage current data at a maximum level of detail, and OLAP systems, which manage historical and aggregated data.
- Data warehouses can use specific design solutions aimed at performance optimization of analysis and report applications.

Finally, it is worth to pay attention to the fact that a few authors use the same terminology to define different concepts. In particular, those authors consider a data warehouse as a repository of integrated and consistent, yet operational, data, while they use a multidimensional representation of data only in data marts. According to our terminology, this "operational view" of data warehouses essentially corresponds to the reconciled data layer in three-layer architectures.

2.3 Three-layer Architecture

In this architecture, the third layer is the reconciled data layer or operational data store. This layer materializes operational data obtained after integrating and cleansing source data. As a result, those data are integrated, consistent, correct, current, and detailed. Fig. 2.3 shows a data warehouse that is not populated from its sources directly, but from reconciled data. The main advantage of the reconciled data layer is that it creates a common reference data model for a whole enterprise. At the same time, it sharply separates the problems of source data extraction and integration from those of data warehouse population. Remarkably, in some cases, the reconciled layer (a.k.a. Operational Data Store) is also directly used to better accomplish some operational tasks, such as producing daily reports that cannot be satisfactorily prepared using the corporate applications, or generating data flows to feed external processes periodically so as to benefit from cleaning and integration. However, reconciled data leads to more redundancy of operational source data. Note that we may assume that even two-layer architectures can have a reconciled layer that is not specifically materialized, but only virtual, because it is defined as a consistent integrated view of operational source data.

2.4 Metadata

In addition to those kinds of data already discussed in section 1.1.1.1, metadata represents a key aspect for the discussed architectures. The prefix "meta-" means "more abstract" (e.g., metarule, metaheuristic, metalanguage, metaknowledge, metamodel, etc.). Prior to come up with a formal definition for metadata, let us formally introduce the data and information concepts. According to the ISO definition, "*data* is a representation of facts, concepts and instructions, done in a formalized manner, useful for

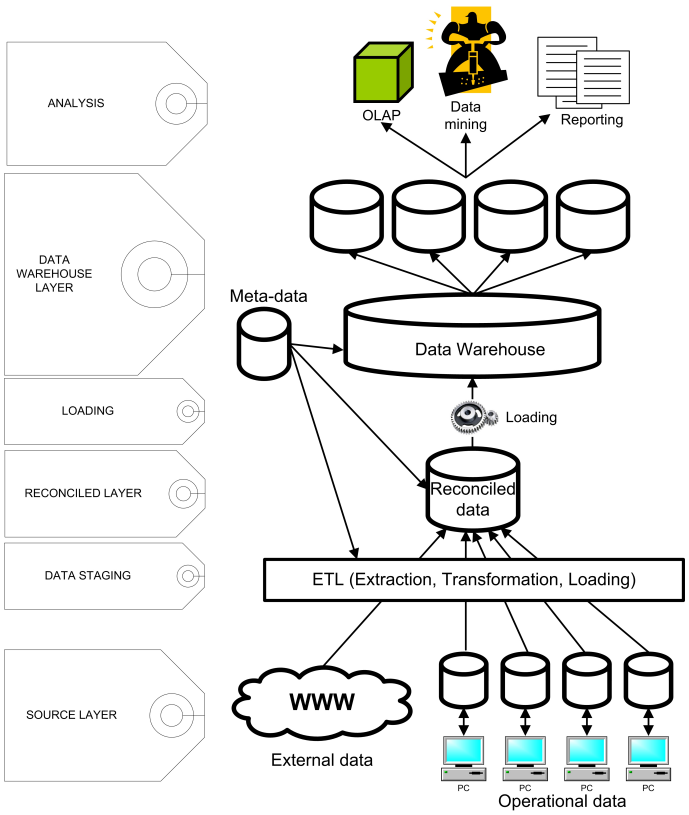


Figure 2.3: Three-layer architecture for a data warehouse system

communication, interpretation and process, by human beings as well as automated means". Information, however, is something more. According to the ISO definition, "*information*, in the processing of data and office machines, is the meaning given to data from the conventional rules used in their representation". The difference can be seen crystal clear with an example. Consider the following datum extracted from a database: 1100. So what information represents this datum? to answer this, it will help to know that this datum is in *binary* format, its type is an *integer*, represents the *age* attribute (in *months*) from a table named *dogs* and this datum is updated every *year* (and last update was in *December*). All these data about the 1100 datum needs to be stored in order to process it as information. This is what we know as metadata, which is applied to the data used to define other data. In short, it is data that allows to interpret data as information. A typical metadata repository is the Relational databases catalog.

In the scope of data warehousing, metadata play an essential role because it specifies source, values, usage, and features of data warehouse data, which is fundamental to come up and justify innovative analysis, but also because they define how data can be changed and processed at every architecture layer (hence, fostering automation). Fig. 2.2 and 2.3 show that the metadata repository is closely connected to the data warehouse. Applications use it intensively to carry out data-staging and analysis tasks. One can classify metadata into two partially overlapping categories. This classification is based on the ways system administrators and end users exploit metadata. System administrators are interested in internal (technical) metadata because it defines data sources, transformation processes, population policies, logical and physical schemas, constraints, and user profiles and permissions. External (business) metadata are relevant to end users. For example, it is about definitions, actualization information, quality standards, units of measure, relevant aggregations, derivation rules and algorithms, etc.

Metadata are stored in a metadata repository which all the other architecture components can access. A tool for metadata management should:

- allow administrators to perform system administration operations, and in particular manage security;
- allow end users to navigate and query metadata;
- use a GUI;
- allow end users to extend metadata;
- allow metadata to be imported/exported into/from other standard tools and formats.

Multimedia Materials

Metadata (en)

Metadata (es)

Chapter 3

OLAP and the Multidimensional Model

OLAP tools are intended to ease information analysis and navigation all through the data warehouse, for extracting relevant knowledge of the organization. This term was first introduced by E.F. Codd in 1993 [CCS93], and it was carefully chosen to confront OLTP. In the context of data warehousing, as depicted in Fig. 3.1, OLAP tools are placed in between the data warehouse and the front-end presentation tools.

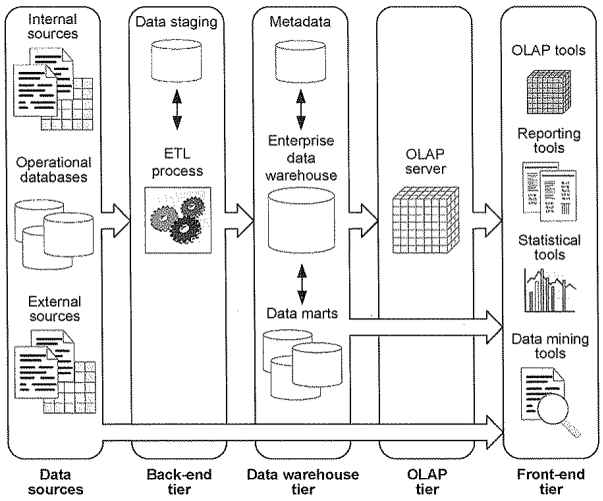


Figure 3.1: OLAP Reference Architecture [VZ14]

OLAP tools are precisely defined by means of the FASMI (*Fast Analysis of Shared Multidimensional Information*) test [Pen08]. According to it, an OLAP tool must provide *Fast* query answering to not frustrate the end-user reasoning; offer *Analysis* tools, implement security and concurrent mechanisms to *Share* the business *Information* from a *Multidimensional* point of view. This last feature is the most important one since OLAP tools are conceived to exploit the data warehouse for analysis tasks based on *multidimensionality*.

We can say that the multidimensionality plays for OLAP the same role as the Relational model for Relational databases. Unfortunately, unlike Relational databases, there is not yet consensus about a standard multidimensional model (among other reasons because major software vendors are not interested to reach such agreement). However, we can nowadays talk about a de facto multidimensional data structure (or multidimensionality), and to some extent, about a de facto multidimensional algebra, which we next present at three different levels: (1) what reality multidimensionality models, (2) how this reality can be represented at the conceptual level, and (3) which are the logical (also physical) alternative representations level.

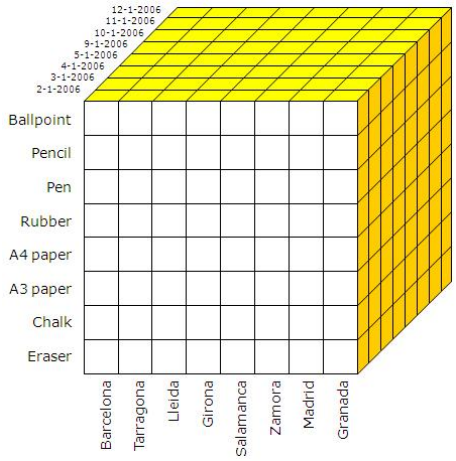


Figure 3.2: Multidimensional view of data

3.1 The Real World: Events and Multidimensionality

The multidimensional point of view is based on the cube metaphor (which is not actually that "real"). Specifically, the multidimensional view of data is distinguished by the *fact* / *dimension* dichotomy, and it is characterized by representing data as if placed in an n-dimensional space, allowing us to easily understand and analyze data in terms of *facts* (the subjects of analysis) and *dimensions* showing the different points of view from where a subject can be analyzed. For instance, Fig. 3.2 depicts `sales` (subject of analysis) of an organization from three different dimensions or perspectives of view (`time`, `product` and `place`). In each cube cell, the sales data would be determined by the corresponding place, product and time. One fact and several dimensions to analyze it give rise to what is known as the *data cube*[1].

This paradigm, unlike SQL and Relational data structure, provides a friendly, easy-to-understand and intuitive visualization of data for non-expert end-users. Importantly, most real-world events recorded nowadays are likely to be analyzed from a multidimensional point of view. An event (a potential fact) is recorded altogether with a set of relevant attributes or features (potential dimensions). For example, consider a `sales` event. We may record the `shop`, `city` and `country` where it was purchased, the `item`, `color`, `size` and `add-ons` selected, the `time` (`hour`, `minute`, `second` and even `millisecond`) and `date` (`day`, `month`, `year`), `payment method`, `price`, `discount applied`, the `customer`, etc. Interestingly, every attribute opens a new perspective of analysis for the `sales` event. In short, multidimensionality is used to model any real-world event consisting of metrics (facts of interest) and a set of descriptive attributes (dimensions) related to these metrics.

More precisely, OLAP functionality is characterized by dynamic multidimensional analysis of consolidated data supporting end-user analytical and navigational activities. Thus, OLAP users must be able to navigate (i.e., query and analyze) data in real-time. The user provides a *navigation path* in which each node (resulting in a data cube) is derived from the previous node in the path (and thus we say that the user *navigates* the data). Each node is transformed into the next one in the path by applying specific multidimensional operators. Most popular multidimensional operators[2] are "roll-up" (increase the aggregation level), "drill-down" (decrease the aggregation level), "slicing and dicing" (specify a single value for one or more members of a dimension) and "pivot" (reorient the multidimensional view). Some works, add "drill-across" (combine data from cubes sharing one or more dimensions) to these

[1] The data cube refers to the placement of factual data in the multidimensional space. And thus, it can be thought as a mathematical function. Nevertheless, nowadays it is rather common also refer to the multidimensional space as the data cube. However, note that, in both cases, it is a language abuse, since the multidimensional space (or the placement of data in the multidimensional space) only gives rise to a cube if three analysis dimensions are considered (in general, we can have many more).

[2] Right now, these operators are intended to be seen as an example. Later in this chapter we will discuss a multidimensional algebra in depth.

basic operations.

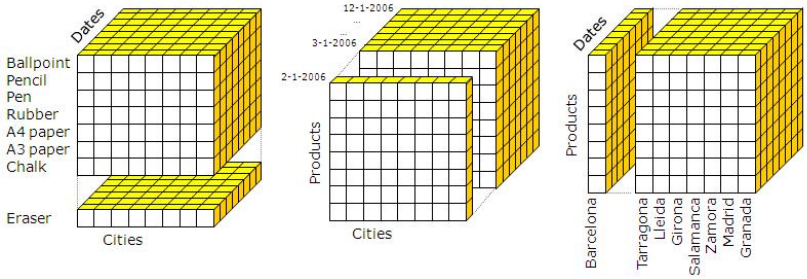


Figure 3.3: Different examples of slice

For example, consider the data cube in Fig. 3.2. Now, the user could decide to slice it by setting constraints (e.g., product = 'Eraser', date = '2-1-2006', city <> 'Barcelona', etc.) over any of the three dimensional axis (see Fig. 3.3) or apply several constraints to more than one axis (see Fig. 3.4). In general, after applying a multidimensional operator, we obtain another cube that we can further navigate with other operations. As a whole, the set of operators applied over the initial cube is what we call the navigation path.

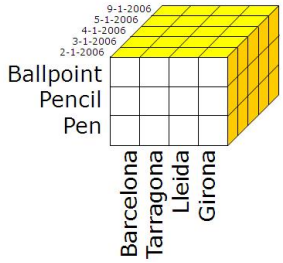


Figure 3.4: Example of dice

As a result, multidimensionality enables analysts, managers, executives and in general those people involved in decision making, to gain insight into data through fast queries and analytical tasks, allowing them to make better decisions.

Roll-up

Sales	January'06	February'06	March'06	April'06
Paper	24	40	15	29
Writing tools	58	40	59	70

Drill-down

Sales		January'06	February'06	March'06	April'06
Paper	Din-A4	24	37	12	27
	Din-A3	0	3	3	2
Writing tools	Ballpoint	15	17	23	20
	Pencil	43	23	36	50

Figure 3.5: An example of roll-up / drill-down over a cube represented in tabular form

3.1.1 Presentation Issues

Although multidimensional events are based on the cube metaphor, any real commercial product hardly shows data cube in 3D. Instead, they flatten these cubes into tabular tables or alternative graphical representations, in what is usually known as BI dashboards (like the one shown in Fig. 1.5). For example, consider Fig. 3.5. There, a *cube* showing sales from January to April 2006 for any kind of products considered as paper or writing tools is shown in tabular form. This cube could be manipulated by the user by means of drill-down (i.e., produce a finer data granularity) and thus, data about specific paper types (such as A3 and A4) and writing tools (such as ballpoints and pencils) are shown (again in tabular form). As a side note, we may get back to the previous data granularity by means of the roll-up operator.

3.2 Conceptual Design: Star Schemas

Lots of efforts have been devoted to multidimensional design, and several methods and approaches have been developed and presented in the literature to support the conceptual design of the data warehouse. Multidimensionality, as it is known today, was first introduced by Kimball in [Kim96], where the author argued about the necessity of an ad hoc designing technique for data warehouses. Multidimensional modeling optimizes the system query performance in contrast to conventional *Entity-Relationship* (ER) models [Che76] (widely used for modeling Relational databases) that are constituted to remove redundancy in the data and optimize OLTP performance.

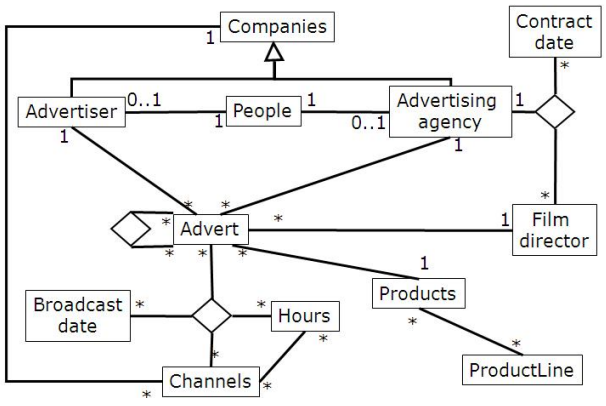


Figure 3.6: A transactional model

Consider Fig. 3.6, which depicts a conceptual transactional UML model (similar to ER). These models were defined to:

- Reduce the amount of redundant data
- Eliminate the need to modify many records because of one modification, very efficient if data change very often

However, it is also well-known that:

- Degrades response time in front of queries (mainly due to the presence of join operators)
- It is easy to make mistakes if the user is not an expert in computer science

These pros and cons suit perfectly operational (typically transactional) systems, but it does not suit decisional systems like data warehouses, which aim at fast answering and easy navigation of data.

As discussed, multidimensionality is based on the fact / dimension dichotomy. **Dimensional concepts** produce the multidimensional space in which the **fact** is placed. What we call here **Dimensional concepts** are those concepts likely to be used as a new analytical perspective, which have traditionally been classified as **dimensions**, **levels** and **descriptors**. Thus, we consider that a **dimension** consists of

a hierarchy of **levels** representing different granularities (or levels of detail) for studying data, and a **level** containing **descriptors** (i.e., **level** attributes, typically descriptive and potentially used to filter or group instances). We denote by **atomic level** the **level** at the bottom of the **dimension** hierarchy (i.e., that of the finest level of detail) and by **All level** the **level** at the top of the hierarchy containing just one instance representing the whole set of instances in the **dimension**. In contrast, a **fact** contains **measures** of analysis (i.e., numerical attributes we want to analyze, typically summarized using some aggregation function). Importantly, note that a **fact** may produce not just one but several different levels of data granularity. Therefore, we say that a certain **granularity** contains individual cells of the same granularity from the same **fact**. A specific **granularity** of data is related to one **level** for each of its associated **dimensions** of analysis. Finally, one **fact** and several **dimensions** for its analysis produce what Kimball called a star schema.

Finally, note that we consider {product × day × city} in Fig. 3.2 to be the multidimensional **base** of the the finest fact granularity level (i.e., that related to the **atomic levels** of each **dimension**, which is also known as the **atomic granularity**). Thus, it means that one value of each one of these **levels** determines one cell (i.e., a sale with its price, discount, etc.). Importantly, this is a relevant feature of multidimensionality. Levels determine factual data or, in other words, they can be depicted as **functional dependencies** (the set of levels determine the fact, and each fact cell has associated a single value from each level). That is the reason why level - fact relationships have 1-* (one-to-many) multiplicities. In the multidimensional model, *-* (many-to-many) relationships are meaningless as they do not preserve the model constraints.

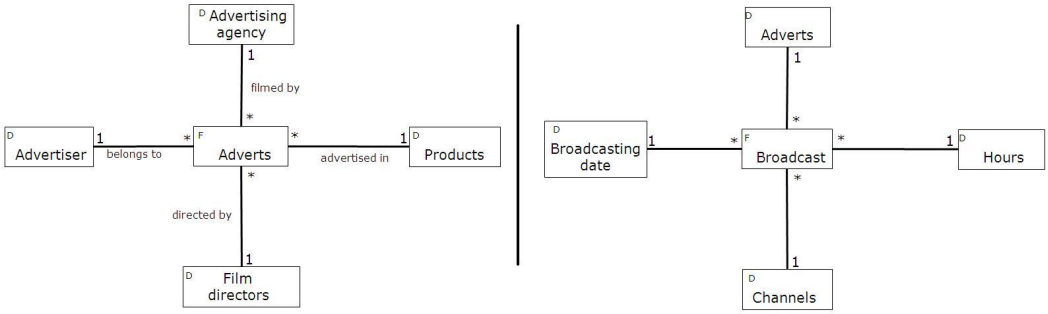


Figure 3.7: Two examples of star-schemas

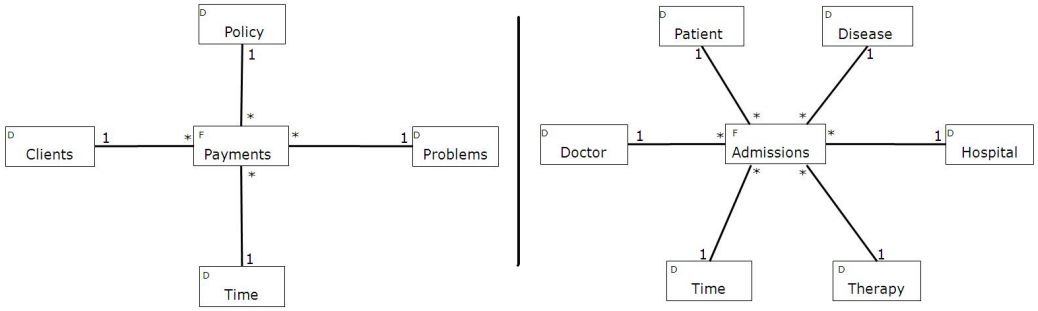


Figure 3.8: Two (more) examples of star-schemas

Recall now the transactional schema used as example in Fig. 3.6. You should be able to distinguish why it cannot be considered a multidimensional schema. Oppositely, check the multidimensional schemas in Figs. 3.7 and 3.8 and see the differences:

- There is a single fact (with some numeric measures or metrics)
- There is a set of dimensions (with descriptive discrete values)

- Only 1:N relationships are considered

Furthermore, they only include data relevant for decision making. Thus, they are simpler and do not contain as much data as transactional schemas. Consequently, they are easier to use and understand, and queries are fast and efficient over such schemas. Note another important property of these schemas: dimensions are represented as a single object. For example, in a Relational implementation (see next section for further details), it would imply there is only one table for all the dimension. Clearly, this is against the second and third Relational normal forms, but denormalization of data is common in data warehousing as we aim to boost querying performance and, in this way, we avoid joins, the most expensive Relational operator. Denormalization is acceptable in such scenarios since the users are not allowed to insert data, and they only query the data warehouse. Thus, since the ETL process is the only responsible to insert data, denormalize such schemas is sound and acceptable (even encouraged).

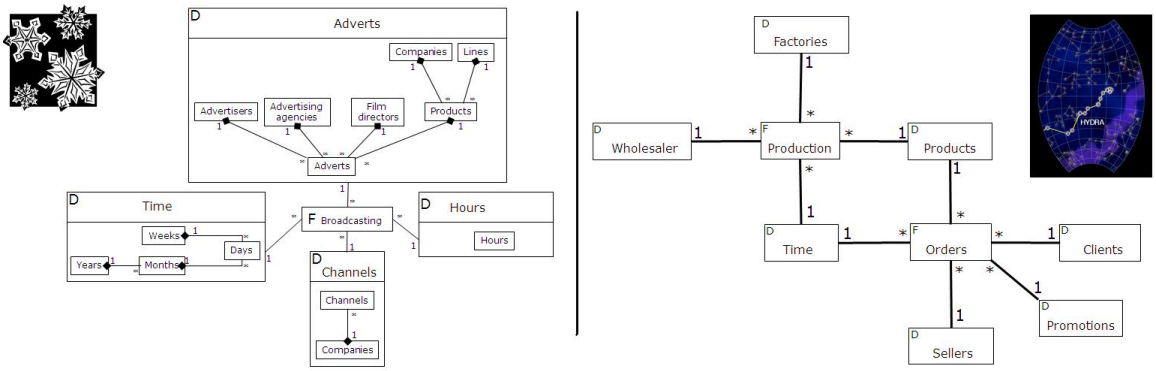


Figure 3.9: Examples of snowflake and constellation schemas

Although the star-schema is the most popular conceptual representation, there are some others such as the snowflake schema and the constellation schema. The first one (see the left-hand side schema on Fig. 3.9) corresponds to a normalized star-schema; i.e., without denormalizing dimensions and hence each concept is explicited separately. Finally, when a schema contains more than one fact, which share dimensions is called a galaxy or constellation (see the right-hand side schema on Fig. 3.9), and facilitates drilling-across (see algebraic operations bellow).

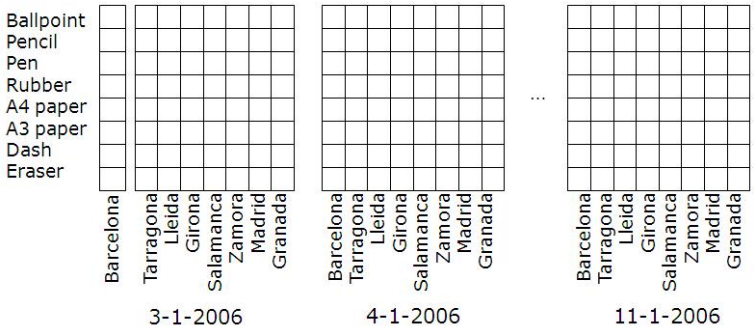


Figure 3.10: Example of a MOLAP tool storing data as arrays

3.3 Logical Design: ROLAP Vs. MOLAP

When implementing our data warehouse, there are two main trends: using the Relational technology or an ad hoc one, giving rise, respectively, to what are known as ROLAP (*Relational On-line Analytical*

Processing) and MOLAP (*Multidimensional On-line Analytical Processing*) architectures. ROLAP maps the multidimensional model over the Relational one (a multidimensional middleware on the top of the Relational database makes this fact transparent for the users), allowing them to take advantage of a well-known and established technology, whereas MOLAP systems are based on an ad hoc logical model that can be used to represent multidimensional data and operations directly. The underlying multidimensional database physically stores data as flatten/serialized arrays (and the access to it is positional), Grid-files, R*-trees or UB-trees, which are among the techniques used for this purpose (see Fig. 3.10).

As consequence, ROLAP tools used to deal (nowadays, this statement is starting to crumble) with larger volumes of data than MOLAP tools (i.e., ad hoc multidimensional solutions), but their performance for query answering and cube browsing is not as good (mainly because Relational technology was conceived for OLTP systems and they tend to generate too many joins when dealing with multidimensionality). Thus, new HOLAP (*Hybrid On-line Analytical Processing*) tools were proposed. HOLAP architecture combines both ROLAP and MOLAP ones trying to obtain the strengths of both approaches, and they usually allow to change from ROLAP to MOLAP and viceversa. Specifically, HOLAP takes advantage of the standardization level and the ability to manage large amounts of data from ROLAP implementations, and the query speed typical of MOLAP systems. HOLAP implies that the largest amount of data should be stored in a Relational DBMS to avoid the problems caused by sparsity, and that a multidimensional system stores only the information users most frequently need to access. If that information is not enough to solve queries, the system will transparently access the part of the data managed by the Relational system.

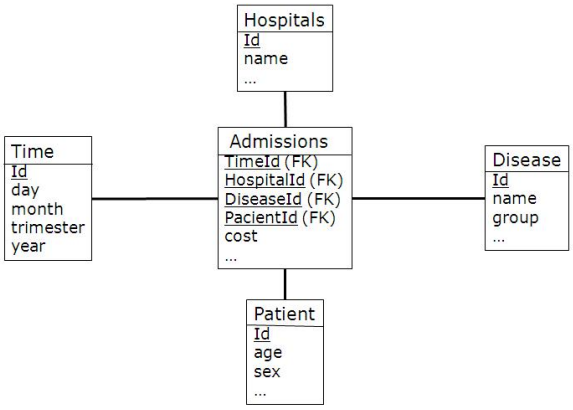


Figure 3.11: Example of a star-join schema corresponding to the right-hand star-schema in Fig. 3.8 (ignoring dimensions "Doctor" and "Therapy" for the sake of simplicity)

Although ROLAP tools have failed to dominate the OLAP market due to its severe limitations (mainly slow query answering) [Pen05], at the beginning, they were the reference architecture. Indeed, Kimball's reference book [KRTR98] presented how a data warehouse should be implemented over a Relational DBMS (*Relational Database Management System*) and how to retrieve data from it. To do so, he introduced for first time the *star-join* (to implement star-schemas) and *snowflake* schemas (to implement the conceptual schemas with the same name). At Relational level, the star schema consists of one table for the fact and one denormalized table for every dimension, with the latter being pointed by *foreign keys* (FK) from the *fact table*, which compose its *primary key* (PK) (see Fig. 3.11). The normalized[3] version of a star schema is a snowflake schema; getting a table for each level with a FK pointing to each of its parents in the dimension hierarchy. Nevertheless, both approaches can be conceptually generalized into a more generic one consisting in partially normalizing the dimension tables according to our needs: completely normalizing each dimension we get a snowflake schema, and not normalizing them at all results in a star schema. In general, normalizing the dimensions requires a very good reason, since it produces a very little gain in the size of dimension and a big loss in the performance of queries (since it

[3] https://youtu.be/SSio_jhAmzg

generates more joins). Normalization was conceived to minimize redundancies that hinder performance in the presence of updates. However, the DW is considered read-only and dimensional data is specially static, since the dynamicity of business is reflected in the fact tables.

To retrieve data from a star-join schema, Kimball presented the SQL cube-query pattern:

```
SELECT l1.ID, ..., ln.ID, [ F( ]c.Measure1[ ) ], ...
FROM Fact c, Level1 l1, ..., Leveln ln
WHERE c.key1=l1.ID AND ...  AND c.keyn=ln.ID [ AND li.attr Op.  K ]
[ GROUP BY l1.ID, ..., ln.ID ]
[ ORDER BY l1.ID, ..., ln.ID ]
```

Hospital	Month	Average Cost
Duran i Reinals	January'06	3300
Duran i Reinals	February'06	4500
Duran i Reinals	...	...
Duran i Reinals	All	4300
Bellvitge	January'06	180
Bellvitge	February'06	300
Bellvitge	...	...
Bellvitge	All	200

Figure 3.12: Example of the output produced by a cube-query

The FROM clause contains the "Fact table" and the "Dimension (or level in case of a snowflake schema) tables". These tables are properly linked in the WHERE clause by "joins" (if a star-schema, only between the fact and dimension tables. In case of snowflake schema, also between dimensional tables) that represent *concept associations*. The WHERE clause also contains logical clauses restricting a specific **level** attribute (i.e., a **descriptor**) to a constant using a comparison operator (used to slice the produced cube). The GROUP BY clause shows the identifiers of the **levels** at which we want to aggregate data. Those columns in the grouping must also be in the SELECT clause to identify the values in the result. Finally, the ORDER BY clause is designed to sort the output of the query. As output, a cube-query will produce a single data cube (i.e., a specific level of data granularity). For example, think of the cube-query necessary to produce the cube (shown in tabular form) in Fig. 3.12, from the star-join schema in Fig. 3.11.

3.3.1 Implementing a Multidimensional Algebra

In a ROLAP implementation, the multidimensional operators are implemented as modifications in the cube-query clauses. To show how it would work, we first introduce a multidimensional algebra and discuss how each operator is implemented over the cube-query.

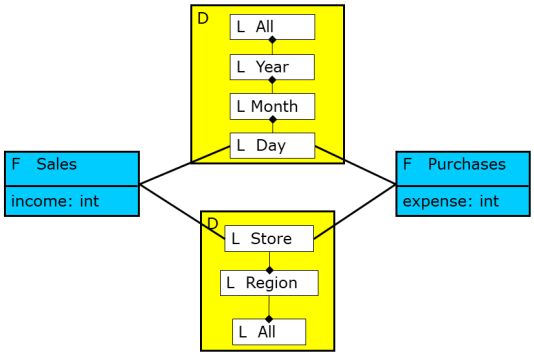


Figure 3.13: Example of conceptual snowflake schema

For the sake of understandability, we present the algebra by means of an example. Consider a snowflake implementation of the conceptual schema depicted in Fig. 3.13. The cube-query that would retrieve the `daily sales` cube is the following:

```
SELECT d.day, p.id, c.name, s.price, s.discount
FROM sales s, day d, product p, city c
WHERE s.product_id = p.id AND s.day = d.day
AND s.city_name = c.name
```

Note that no grouping is needed as we are just retrieving atomic data. Next, we show how this cube-query would be modified by each multidimensional operator next introduced (we suggest the reader to follow Fig. 3.14 to grasp the idea behind each operator):

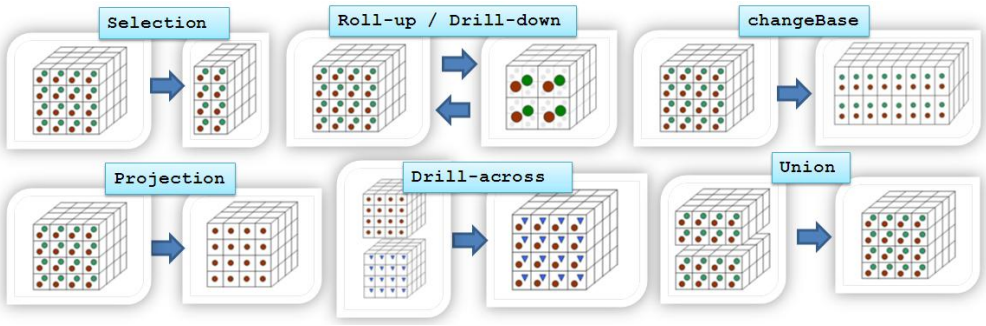


Figure 3.14: Conceptual representation of the multidimensional operators

- **Selection or Dice**: By means of a logic predicate over the dimension attributes, this operation allows users to choose the subset of points of interest out of the whole n-dimensional space (remember to check Fig. 3.14).

Figure 3.15: Exemplification of an OLAP navigation path translation into SQL queries

In SQL, it means to *and* the corresponding comparison clause to the cube-query WHERE clause. For example, consider the atomic cube-query presented as example. If we want to analyze the `sales` data regarding to the `city` of `Barcelona`, we must perform a **selection** over the `city` dimension (see Fig. 3.15).

- **Roll-up**: Also called "Drill-up", it groups cells in a Cube based on an aggregation hierarchy. This operation modifies the granularity of data by means of a many-to-one relationship which relates instances of two levels in the same dimension. For example, it is possible to roll-up monthly sales into yearly sales moving from "Month" to "Year" level along the temporal dimension.

  In SQL, it entails to replace the identifiers of the level from where we **roll-up** with those of the level that we **roll-up** to. Thus, the SELECT, GROUP BY and ORDER BY clauses must be modified accordingly. Measures in the SELECT clause must also be summarized using an aggregation function. In our example (see Fig. 3.15), we perform two different **roll-ups**: on the one hand, we **roll-up** from `product_id` to the `All` **level**. On the other hand, we **roll-up** from `city` to `country`. Note that the `country` table is added to the FROM clause, and we replace the `city` identifier with that of the `country` level in the SELECT, GROUP BY and ORDER BY clauses. Finally, we add the proper links in the WHERE clause. About **rolling-up** from `product` to the `All` level, note that it is equivalent to remove both the `product` identifiers and its links.

- **Drill-down:** This is the counterpart of Roll-up. Thus, it removes the effect of that operation by going down through an aggregation hierarchy, and showing more detailed data.

  In SQL, Drill-down can only be performed by undoing (i.e, changes introduced in the cube-query) the Roll-up operation.

- **ChangeBase**: This operation reallocates exactly the same instances of a cube into a new n-dimensional space with exactly the same number of points. Actually, it allows two different kinds of changes in the space: rearranging the multidimensional space by reordering the dimensions, interchanging rows and columns in the tabular representation (this is also known as pivoting), or adding/removing dimensions to/from the space.

  In SQL it can be performed in two different ways. If we reorder the base (i.e., when "pivoting"), we just need to reorder the identifiers in the ORDER BY and SELECT clauses. But if **changing the base**, we need to add the new level tables to the FROM and the corresponding links to the WHERE clause. Moreover, identifiers in the SELECT, ORDER BY and GROUP BY clauses must be replaced appropriately. Following with the same example shown in Fig. 3.15, we can change from {day × country × All} to {day × country}. Note that both bases are conceptually related by means of a one-to-one relationship. Specifically, this case typically applies when dropping a dimension (i.e., **rolling-up** to its `All` level and then **changing the base**). We **roll-up** to the `All` for representing the whole dimensions instances as a single one and therefore, producing the following base: {day × country × 1}. Now, we can **changeBase** to {day × country} without introducing aggregation problems (since we **changeBase** through a one-to-one relationship).

- **Drill-across**: This operation changes the subject of analysis of the cube, by showing measures regarding a new fact. The n-dimensional space remains exactly the same, only the data placed in it change so that new measures can be analyzed. For example, if the cube contains data about sales, this operation can be used to analyze data regarding stock using the same dimensions.

  In SQL, we must add a new fact table to the FROM clause, its measures to the SELECT, and the corresponding links to the WHERE clause. In general, if we are not using any semantic relationship, a new fact table can always be added to the FROM clause if fact tables share the same base. In our example, suppose that we have a `stock` cube sharing the same dimensions as the `sales` cube. Then, we could **drill-across** to the `stock` cube and show both the `stock` and `sales` measures (see Fig. 3.15).

- **Projection**: It selects a subset of measures from those available in the cube.

  In SQL it entails to remove measures from the SELECT clause. Following our example, we can remove the `discount` measure by projecting the `stock` and `price` measures.

- **Set operations**: These operations allow users to operate two cubes defined over the same n-dimensional space. Usually, Union, Difference and Intersection are considered.

  In this document, we will focus on Union. Thus, in SQL, we unite the FROM and WHERE clauses of both SQL queries and finally, we *or* the **selection** conditions in the WHERE clauses. Importantly,

note that we can only **union** queries over the same fact table. Intuitively, it means that, in the multidimensional model, the **union** is used to undo **selections**. We can unite our example query to one identical but querying for data concerning `Lleida` instead of `Barcelona`.

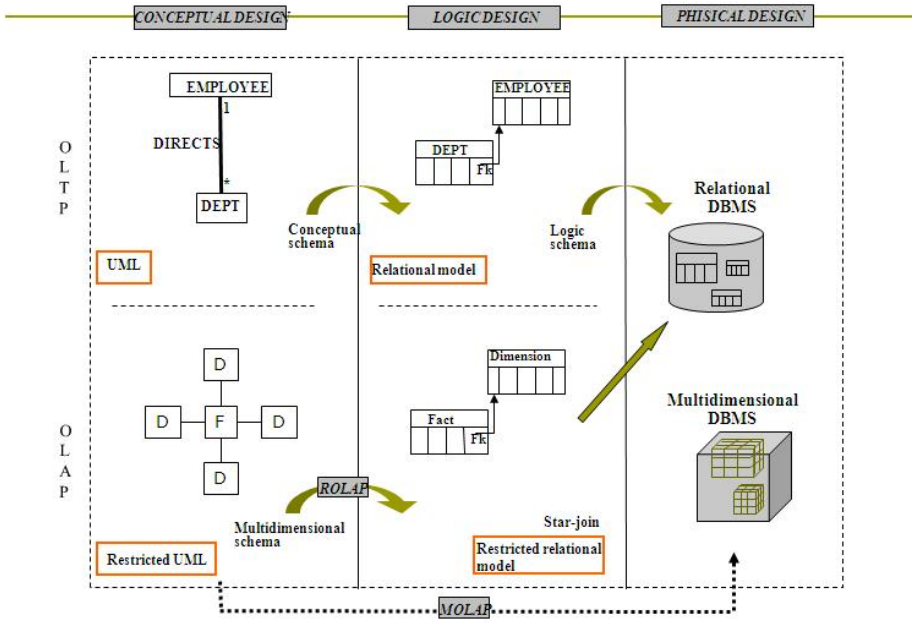


Figure 3.16: Sketch summarizing the main design differences between a transactional and a decisional system

3.4 SQL Grouping Sets

Computing aggregate data is a basic feature for OLAP (as well as for other tools). For this reason, the standard SQL-99 added 3 new modifiers to the GROUP BY clause in order to support and enhance advanced aggregate computation. In this document we discuss these modifiers as well as their usefulness to implement star-join (i.e., ROLAP) implementations. In short, and as you will realize after reading this document, these modifiers tune up the GROUP BY clause syntax (without changing its behaviour).

The new GROUP BY modifiers are the following: GROUPING SETS, ROLLUP and CUBE. Although their names may ring a bell to multidimensional concepts already introduced in this course, **do not get mislead by their names**, which are not really appropriate according to the multidimensional theory.

Let us first introduce a traditional case study, upon which we will introduce these new keywords. Consider the following star-schema in Figure 3.17:

It is a simple star-schema, with 3 dimensions (product, time and place). Suppose now a star-join logical implementation and assume we aim at showing data (i.e., a data cube) at Item, Month and Region granularity. As presented earlier in this course, the cube-query pattern is intended to retrieve data from a ROLAP star-schema implementation. To do so, the cube-query must retrieve the Sales atomic data and perform two roll-ups (one over the time dimension from Day to Month and the other one over the Place dimension from City to Region). The result would be the following:

```
SELECT d1.itemName, d2.region, d3.monthYear, SUM(f.items)
FROM Sales f, Product d1, Place d2, Time d3
WHERE f.IDProduct=d1.ID AND f.IDPlace=d2.ID AND f.IDTime=d3.ID
      AND d1.itemName IN (`Ballpoint','Rubber') AND d2.region='Catalonia'
      AND d3.monthYear IN (`January2002','February2002')
```

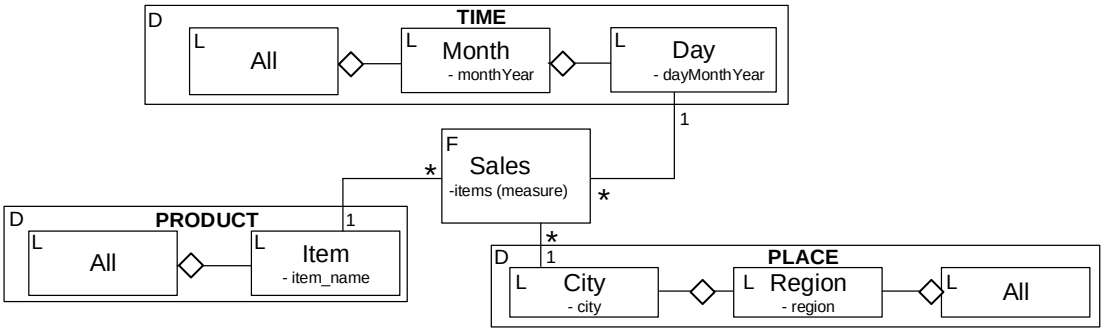


Figure 3.17: Sales star schema

```
GROUP BY d1.itemName, d2.region, d3.monthYear
ORDER BY d1.itemName, d2.region, d3.monthYear
```

For the sake of comprehension, the reader will note three slicers over each dimension. The reason is to restrict the amount of values to be shown in the upcoming figures.

3.4.1 GROUPING SETS

Typically, the users do not want to just visualize data at a specific aggregation level (i.e., at a certain granularity), but *totals* are of their interest. These aggregations are depicted in a kind of table known as *cross-tab*. For example:

Sales	Catalonia		
	January02	February02	Total
Ballpoint	275827	290918	566745
Rubber	784172	918012	1702184
Total	1059999	1208930	2268929

This table is not a proper cube because it mixes data cells from different granularities or cubes (each granularity represented by a different colour). Note, however, that this table can be seen as the union of four different cubes (i.e., four different granularity levels). For example, the yellow cells represent the data cube obtained at the {Item, Region and Month} granularity. In other words, they represent the Sales fact at its atomic granularity. The other cells are produced by aggregating data to the All level of some dimensions. Specifically, green cells represent the cube at the {Month, Region, All} granularity level (i.e., a roll-up over the Item dimension has been performed to the All level), the red ones at {All, Region, Item} granularity and the blue ones at {All, Region, All}.

Indeed, in the same way we can define a function piecewise, we can do it with a cross-tab by uniting cube pieces. For example:

$$\text{Sales} \oplus \textit{Roll-up}_{Time::All}(\text{Sales}) \oplus \textit{Roll-up}_{Items::All}(\text{Sales}) \oplus \textit{Roll-up}_{Items::All,Time::All}(\text{Sales})$$

Where $\oplus$ denotes a cube composition table-wise. Now, go back to the previous cube-query introduced and realize that that query only retrieves the four yellow cells. To obtain the other cells we need to union four queries as shown below:

```
SELECT d1.itemName, d2.region, d3.monthYear, SUM(fact.items)
 FROM Sales fact, Product d1, Place d2, Time d3
 WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
```

```
 AND d1.itemName IN (`Ballpoint','Rubber') AND d2.region='Catalonia'
 AND d3.monthYear IN (`January2002','February2002')
 GROUP BY d1.itemName, d2.region, d3.monthYear
UNION
SELECT d1.itemName, d2.region, `Total', SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
 AND d1.itemName IN (`Ballpoint','Rubber') AND d2.region='Catalonia'
 AND d3.monthYear IN (`January2002','February2002')
GROUP BY d1.itemName, d2.region
UNION
SELECT `Total', d2.region, d3.monthYear, SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
 AND d1.itemName IN (`Ballpoint','Rubber') AND d2.region='Catalonia'
 AND d3.monthYear IN (`January2002','February2002')
GROUP BY d2.region, d3.monthYear
UNION
SELECT `Total', d2.region, `Total', SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
 AND d1.itemName IN (`Ballpoint','Rubber') AND d2.region='Catalonia'
 AND d3.monthYear IN (`January2002','February2002')
GROUP BY d2.region
ORDER BY d1.itemName, d2.region, d3.monthYear;
```

Check carefully each united piece. They are almost identical. Indeed, formally, we are only changing the GROUP BY clause to obtain the desired granularity. Furthermore, if we were interested in computing the totals for all three dimensions instead of two, we would need 7 unions; i.e., 8 different SQL queries at different granularities only changing their GROUP BY clause. In other words, in addition to those granularities computed in the example introduced we should compute the {Month, All, Item}, {Month, All, All} and {All, All, Item} granularities.

Figure 3.18 sketches the eight cubes to union (Sales per Day, Product and City; per Day and Product; per Day and City; per Product and City; per Day; per Product; per City; and the overall total).

It is easy to realize that the amount of unions to perform grows at an exponential rate with regard to the number of dimensions. Fortunately, the SQL'99 standard provides specific syntax to save us time. As shown in the next query, we can use the GROUPING SETS keyword in the GROUP BY clause to produce the desired granularities (in brackets after the keyword). In this way the other query clauses are written only once:

```
SELECT d1.itemName, d2.region, d3.monthYear, SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
      AND d1.itemName IN (`Ballpoint', 'Rubber') AND d2.region='Catalonia'
      AND d3.monthYear IN (`January2002','February2002)
GROUP BY GROUPING SETS ((d1.itemName, d2.region, d3.monthYear),
                        (d1.itemName, d2.region),
                        (d2.region, d3.monthYear),
```

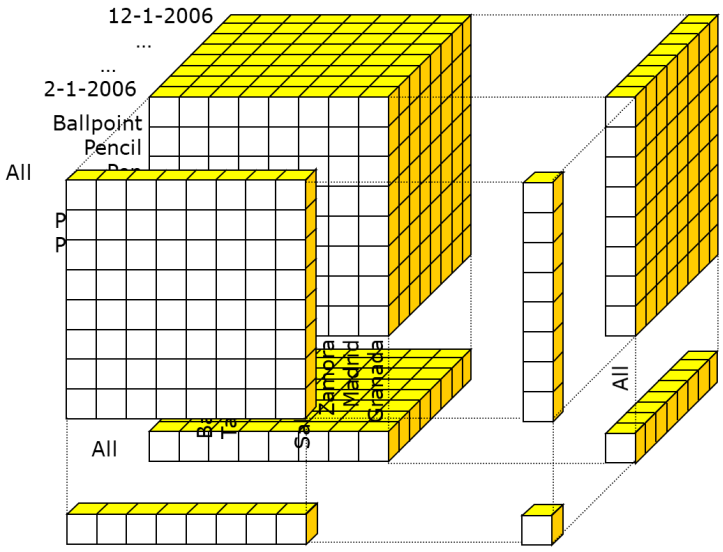


Figure 3.18: Cubes

```
                                    (d2.region))
ORDER BY d1.itemName, d2.region, d3.monthYear;
```

The query output would be the following:

itemName	Region	monthYear	items
Ballpoint	Catalonia	January02	275827
Ballpoint	Catalonia	February02	290918
Ballpoint	Catalonia	NULL	566745
Rubber	Catalonia	January02	784172
Rubber	Catalonia	February02	918012
Rubber	Catalonia	NULL	1702184
NULL	Catalonia	January02	1059999
NULL	Catalonia	February02	1208930
NULL	Catalonia	NULL	2268929

Besides the visualization problem (which the corresponding exploitation tool is responsible for), pay attention to the NULL values. What is their meaning in this context? Is it 'unknown', maybe 'not applicable'? Well, none of them. Indeed, it is a new meaning introduced in the SQL'99 standard which means 'total' (i.e., it represents the All value for that dimension).

But as you might have thought, how can we distinguish the meaning between different NULL values? In other words, given a NULL value, how can we know if this is the result of aggregating data or an unknown value? To answer this question, the standard introduces the GROUPING function. This function takes as parameter an attribute and it returns '1' if it represents a total (i.e., produced by an aggregation). Otherwise, it returns '0'. Thus, you might need to use this function for presentation issues and, for example, rewrite the previous query to replace the NULL values by the total word (note that d2.region appears in all four aggregations and thus, it never holds for the total meaning):

```
SELECT
CASE WHEN GROUPING(d1.itemName)=1 THEN `TotalOfBallpointAndRubber'
     ELSE d1.itemName,
     d2.region,
CASE WHEN GROUPING(d3.monthYear)=1 THEN `TotalOfJanuaryAndFebruary'
     ELSE d3.monthYear,
```

```
     SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
AND d1.itemName IN (`Ballpoint', 'Rubber') AND d2.region='Catalonia'
AND d3.monthYear IN (`January2002','February2002)
GROUP BY GROUPING SETS ((d1.itemName, d2.region, d3.monthYear),
                        (d1.itemName, d2.region),
                        (d2.region, d3.monthYear),
                        (d2.region))
ORDER BY d1.itemName, d2.region, d3.monthYear;
```

In this case, the query output would be the following:

itemName	Region	monthYear	items
Ballpoint	Catalonia	January02	275827
Ballpoint	Catalonia	February02	290918
Ballpoint	Catalonia	TotalOfJanuaryAndFebruary	566745
Rubber	Catalonia	January02	784172
Rubber	Catalonia	February02	918012
Rubber	Catalonia	TotalOfJanuaryAndFebruary	1702184
TotalOfBallpointAndRubber	Catalonia	January02	1059999
TotalOfBallpointAndRubber	Catalonia	February02	1208930
TotalOfBallpointAndRubber	Catalonia	TotalOfJanuaryAndFebruary	2268929

3.4.2 ROLLUP

As previously discussed, the number of totals grows at an exponential rate regarding the number of dimensions in the cube. As a consequence, even if we can save writing the other query clauses by using the GROUPING SETS modifier, it can still happen to be unbearable in some cases. In other words, the GROUPING SETS avoids writing the same query n times, but we still have to write all the attribute combinations needed to produce the desired granularities.

To facilitate even more computing aggregations, the SQL'99 standard introduced the ROLLUP keyword. Given an attribute set, it computes all the aggregations by disregarding, on each grouping, the right-most attribute in the set. Thus, it is not a set in the mathematical sense, as order does matter. Consider the previous query that used the GROUPING SETS modifier and how it can be rewritten using the ROLLUP keyword (pay attention to the attribute order in the GROUP BY and ORDER BY clauses):

```
SELECT d1.itemName, d2.region, d3.monthYear, SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
      AND d1.itemName IN (`Ballpoint', 'Rubber') AND d2.region='Catalonia'
      AND d3.monthYear IN (`January2002','February2002')
GROUP BY ROLLUP (d2.region, d1.itemName, d3.monthYear);
ORDER BY d2.region, d3.monthYear, d1.itemName;
```

The query output would be:

itemName	region	monthYear	items
Ballpoint	Catalonia	January02	275827
Rubber	Catalonia	January02	784172
Ballpoint	Catalonia	February02	290918
Rubber	Catalonia	February02	918012
Ballpoint	Catalonia	NULL	566745
Rubber	Catalonia	NULL	1702184
NULL	Catalonia	NULL	2268929
NULL	NULL	NULL	2268929

The first four rows correspond to the "GROUP BY d2.region, d1.itemName, d3.monthYear"; next two to the "GROUP BY d2.region, d1.itemName"; next one to the "GROUP BY d2.region"; and the last one to the "GROUP BY ()" (note that from the SQL'99 standard on it is allowed to write GROUP BY(); i.e., with the empty list). If we happen to have a single value for d2.region (e.g., Catalonia), the measure value in the two last rows happens to be identical.

Realize, accordingly, that GROUP BY ROLLUP (a_1,..,a_n) corresponds to:

```
GROUP BY GROUPING SETS ((a_1,...,a_n),
                        (a_1,...,a_{n-1}),
                        ...
                        (a_1),
                        ())
```

Importantly, note these two relevant features. Firstly, the order you write the attributes in the ROLLUP list does matter (it determines the aggregations to be performed). However, the attribute order specified in the ORDER BY clause does not affect the query result (only how it is presented to the user). Secondly, we can avoid producing the last row by fixing the corresponding dimension to a single value (in our case, it is fixed to Catalonia, check the slicer over the Place dimension in the query) and forcing this value to be present in all aggregations computed (i.e., placing it in the GROUP BY but out of the ROLLUP expression):

```
SELECT d1.itemName, d2.region, d3.monthYear, SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
      AND d1.itemName IN (`Ballpoint', 'Rubber') AND d2.region='Catalonia'
      AND d3.monthYear IN (`January2002','February2002')
GROUP BY d2.region, ROLLUP (d1.itemName, d3.monthYear);
ORDER BY d2.region, d3.monthYear, d1.itemName;
```

This query output is exactly the same as the one presented before but without the last row.

Note that we can rewrite the GROUPING SETS exemplifying query presented in previous section (which contained four aggregations) as follows:

```
SELECT d1.itemName, d2.region, d3.monthYear, SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
      AND d1.itemName IN (`Ballpoint', 'Rubber') AND d2.region='Catalonia'
      AND d3.monthYear IN (`January2002','February2002)
GROUP BY GROUPING SETS ((d2.region, ROLLUP (d1.itemName, d3.monthYear)),
                        (d2.region, d3.monthYear))
ORDER BY d1.itemName, d2.region, d3.monthYear;
```

3.4.3 CUBE

ROLLUP saves us from writing down all attribute combinations in order to produce the desired granularities. However, we are still forced to write some of them (check the previous example). But it is

possible to produce all the combinations needed for the GROUPING SETS exemplifying query by using the CUBE keyword. Check the following query; it produces the 9 cells of that example in a more concise way:

```
SELECT d1.itemName, d2.region, d3.monthYear, SUM(fact.items)
FROM Sales fact, Product d1, Place d2, Time d3
WHERE fact.IDProduct=d1.ID AND fact.IDPlace=d2.ID AND fact.IDTime=d3.ID
      AND d1.itemName IN (`Ballpoint', 'Rubber') AND d2.region='Catalonia'
      AND d3.monthYear IN (`January2002','February2002)
GROUP BY d2.region, CUBE (d1.itemName, d3.monthYear);
ORDER BY d1.itemName, d2.region, d3.monthYear;
```

Thus, GROUP BY CUBE (a, b, c) corresponds to:

```
GROUP BY GROUPING SETS ((a,b,c),
                        (a,b),
                        (a,c),
                        (b,c),
                        (a),
                        (b),
                        (c),
                        ())
```

In addition, CUBE and ROLLUP can be combined in the same GROUP BY expression to obtain the desired attribute combinations. For example:

GROUP BY CUBE(a,b), ROLLUP(c,d) corresponds to:

```
GROUP BY GROUPING SETS ((a,b,c,d),
                        (a,b,c),
                        (a,b),
                        (a,c,d),
                        (a,c),
                        (a),
                        (b,c,d),
                        (b,c),
                        (b),
                        (c,d),
                        (c),
                        ())
```

Indeed, the SQL'99 standard allows combining CUBE, ROLLUP and GROUPING SETS to a certain extent. We suggest to practice with a toy example and realize which combinations make sense and which do not.

3.4.4 Conclusions

The CUBE, ROLLUP and GROUPING SETS keywords were introduced in the SQL'99 standard as modifiers for the GROUP BY clause. They were intended to facilitate aggregation computations and thus, they can be considered as syntactic sugar. **However, they are something else than pure syntactic sugar, as they do improve the system performance since the query optimizer receives valuable additional information about the queries to be carried out**.

3.5 Final Discussion

All in all, the design steps to undertake when designing a OLTP or a OLAP system show some inherent differences, since they must consider different premises. Fig. 3.16 sums up all the discussion introduced in this chapter:

- A transactional conceptual schema has no further restrictions than those of the application domain. For example, it can be modeled using any UML feature. However, multidimensional schemas are a simplified version, denoted by the star-shaped schemas.
- OLTP systems are traditionally implemented using the Relational technology, which has been proven to suit their necessities. However, a multidimensional schema can be either implemented via ROLAP or MOLAP approaches.

Multimedia Materials

Motivation and Definition (en) 

Motivation and Definition (ca) 

Real World (en) 

Real World (ca) 

Conceptual World (en) 

Conceptual World (ca) 

Representations World (en) 

Representations World (ca) 

SQL Transformations with Multidimensional Algebra (en) 

SQL Transformations with Multidimensional Algebra (ca) 

Advanced Multidimensional Design (en) 

Advanced Multidimensional Design (ca) 

Summarizability Conditions (en) 

Summarizability Conditions (ca) 

Grouping Sets (en) 

Chapter 4

Query optimization

A Database Management System (DBMS) is much more than a file system (see [GUW09]). Thus, before looking at the details of the query optimization process, we will see a functional architecture of a DBMS. A functional architecture is that representing the theoretical components of the system as well as their interaction. Concrete implementations do not necessarily follow such architecture, since they are more concerned with performance, while a functional architecture is defined for communication, teaching and learning purposes.

Indeed, a DBMS offers many different functionalities, but maybe the most relevant to us is query processing, which includes views, access control, constraint checking and, last but not least, optimization. The latter functionality consists on deciding the best way to access the requested data (a.k.a. access plan). Once it has been decided, the plan is passed through the execution manager that allows to handle a set of operations as in a coordinated way. Finally, the data manager helps to avoid the disk bottleneck by efficiently moving data from disk to memory and vice-versa, guaranteeing that no data is lost in case of system failure.

4.1 Functional architecture of a DBMS

In this section, we analyse the different components of a DBMS. A component is a piece of software in charge of a given functionality. It is in this sense that we introduce a functional architecture (i.e., the organization and interaction of the components taking care of the different functionalities of the DBMS).

Figure 4.1 shows that we have to manage persistent (a.k.a. disk) as well as volatile (a.k.a. memory) storage systems. This implies the implementation of some recovery mechanisms to guarantee that both are always synchronized and data is not lost in case of system or media failures. On top of this, the presence of concurrent users forces to schedule the different operations to avoid interferences between them. Both functionalities are necessary to guarantee that some operations of a user in a session can be managed together into a logic unit called transaction. However, as explained before, the most important components of a DBMS are the query manager and the execution manager. The former provides many functionalities, namely security management, view resolution, constraint checking and query optimization, and generates a query plan that the execution manager executes.

We must take into account that the stable, as well as the volatile storage may be distributed (if so, different fragments of data can be located at different sites). Also many processors could be available (in which case, the query optimizer has to take into account the possibility of parallelizing the execution). Obviously, this would affect some parts of this architecture, mainly depending on the heterogeneities we find in the characteristics of the storage systems we use and the semantics of the data stored. However, we are not going into details in this chapter, because we focus on centralized databases.[1]

[1] Parallel and distributed computing dramatically influences the different components of this architecture.

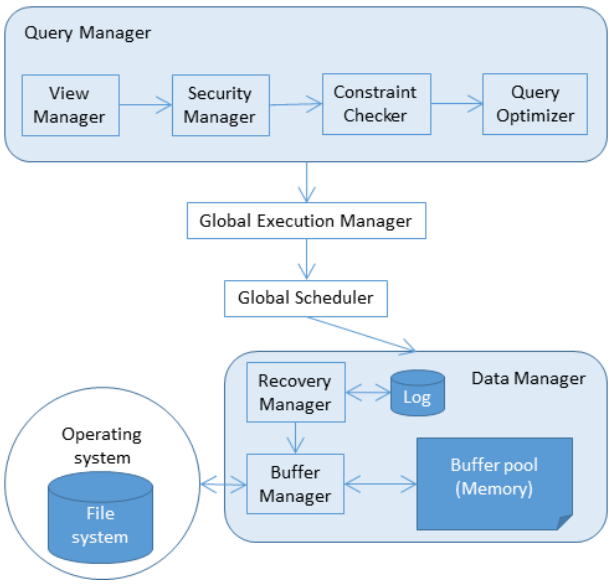


Figure 4.1: Functional architecture of a DBMS

4.1.1 Query manager

The query manager is the module of the DBMS in charge of transforming a declarative SQL query into an ordered set of steps (i.e., a procedural description usually in terms of algebraic operations). This transformation is even more difficult taking into account that, following the ANSI/SPARC architecture, DBMSs must provide views to deal with semantic relativism and logical independence (see Chapter 5). It is also relevant to note that security and constraints have to be taken into account.

4.1.1.1 View manager

From the point of view of the final user, there is no difference on querying data in a view or a table. However, dealing with views is not an easy problem and we are not going to go into details in this chapter. Nevertheless, we want just to briefly show in this section the difficulties it raises.

First of all, we must take into account that data in the view can be physically stored (a.k.a. *materialized*) or not, which raises new difficulties. If data in the view is not physically stored, in order to transform the query over the views into a query over the source tables, we must replace the view name in the user query by its definition (this is known as *view expansion*). In some cases, it is more efficient to instruct the DBMS to calculate the view result and store it waiting for the queries. However, if we do it, we have to be able to transform an arbitrary query over the tables into a query over the available materialized views (this is known as *query rewriting*), which is kind of the opposite to view expansion (in the sense that we have to identify the view definition in the user query and replace it by the view name). If we are able to rewrite a query, we still have to decide whether it is worth rewriting it or not, i.e., the redundant data can be used to improve the performance of the query (this is known as *answering queries using views*).

Finally, updating data in the presence of views is also more difficult. Firstly, we would like to allow users to express not only queries but also updates in terms of views (this is known as *update through views*), which is only possible in few cases. Secondly, if views are materialized, changes in the tables are, potentially, propagated to the views (this is known as *view updating*).

4.1.1.2 Security manager

Ethics as well as legal issues raise the need to control access to data. We cannot allow any user to query or modify all data in our database. This module is in charge of defining user privileges, and once this is

done, validate user statements by checking whether they are allowed to perform the associated action or not.

4.1.1.3 Constraint checker

Another important aspect, also sometimes required by law, is guaranteeing the integrity of data. The database designer defines constraints over the schema that later have to be enforced so that user modifications of data do not violate it. Despite it is not evident, its theoretical background is tightly related to that of view management. Nevertheless, we are not going to elaborate on the problems related to constraint checking implementation, because in a DW final users do not directly/manually modify data, but this is done through established ETL processes run off-line.

4.1.1.4 Query optimizer

Given an SQL sentence, the objective of query processing is to retrieve the corresponding data. However, since SQL is a declarative language, users state the data they want to obtain but not the way it must be retrieved. Thus, clearly, the most important and complex part of the query manager is the optimizer, because it tries to find the best way to execute user statements, and its behaviour will directly impact the performance of the system. It follows the following steps:

1. Validation
   (a) Check syntax
   (b) Check permissions
   (c) Expand views
   (d) Check table schema
2. Optimization
   (a) Semantic
   (b) Syntactic
   (c) Physical
3. Evaluation (i.e., disk access)

We should remember that it has three components, namely semantic, syntactic and physical optimizers (see details in Section 4.2).

4.1.2 Execution manager

The query optimizer decomposes the query in a set of atomic operations (mostly corresponding to those in Relational algebra). It is the task of this component to coordinate the execution of such operations, step by step. In distributed environments, it is also the responsibility of this component to assign the execution of each operation to a given site (in this case, there is also a "local" execution manager, not depicted in the figure).

4.1.3 Scheduler

As you know, many users (up to tens or hundreds of thousands) can work concurrently on a database. In such case, it is quite likely that they want to access not only the same table, but exactly the same column and row. If so, they can interfere the task one another. The DBMS must provide some mechanisms to deal with this problem. Roughly speaking, the way to do it is to restrict the execution order of the operations (i.e., reads, writes, commits and aborts) of the different users. On getting a command, the scheduler can either pass it directly to the data manager, queue it waiting for the appropriate time to be executed, or definitely cancel it (resulting in the abortion of its transaction). The most basic and commonly used mechanism to avoid interferences is Shared-eXclusive locking (see [BHG87]).

4.1.4 Data manager

It is well known that memory storage is much faster than disk storage (usually thousands of times, but maybe even more). Unfortunately, it is volatile and much more expensive (hence scarce). Firstly, being expensive means that its size is limited, and it can only contain a small part of the database. On the other hand, being volatile means that on switching off the server or just rebooting it, we would lose our data. It is the task of the data manager to benefit from both kinds of storage (memory and disk) while smoothing their weaknesses. This has a specific subcomponent in charge of moving data from disk to memory (a.k.a. *fetch*) and from memory to disk (a.k.a. *flush*) to satisfy the requests it gets from other components. Let's briefly explain now its two components.

4.1.4.1 Buffer manager

Data have to be moved from disk to memory to make them accessible to other components of the DBMS. Sooner or later, if data have not been modified, they are simply removed from memory to make room for other data. However, when they have been modified, data have to be moved from memory to disk in order to avoid losing that modification.

The simplest way to manage this is known as *write through*.[2] Unfortunately, this is quite inefficient, because the disk (which is a slow component) becomes a bottleneck for the whole system. We should note that it would be much more efficient to leave a piece of data in memory waiting for several modifications, and then make all of them persistent at the same time with only one disk writing (a.k.a. *flush*). Thus, we will have a buffer pool to keep data temporally in memory expecting many modifications in a short period of time.[3] It is also important to note that the interface to the buffer manager is always through the recovery manager (i.e., other components than this cannot gain direct access to it).

4.1.4.2 Recovery manager

Waiting for many modifications before writing data into disk may result in losing some user requests in case of power failure. Imagine a situation where a given user executes a statement modifying tuple t_1, the system confirms the execution, but it does not write the data into disk waiting for other modifications of tuples in the same block;[4] unfortunately, before more modifications arrive there is a power failure. In case of system failure, all the user would have what is stored into disk. If the DBMS would not implement some safety mechanism, that modification of t_1 would be lost. It is the task of this component to avoid such loss. Moreover, it is also the task of this component to undo all changes done during a rolled back transaction.[5] As it happens with other components, doing this in a distributed environment is much more difficult than in a centralized one.

Not only system failures must be foreseen, but also media failures. For example, in case of a disk failure, it will also be the responsibility of the recovery manager to provide the means to recover all data in it. Remember that the "Durability" property of centralized ACID transactions states that once a modification is committed it cannot be lost under any circumstances.

4.2 Query optimizer

Given a query, there are many strategies that a DBMS can follow to process it and produce its answer. All of them are equivalent in terms of their final output, but vary in their cost, that is, the amount of time and resources that they need to run. This cost difference can be several orders of magnitude large. Thus, all DBMSs have a module that examines "all" alternatives and chooses the plan that needs the least amount of time or resources (see [Ioa96]). This module is called the *query optimizer*.

[2] Write through means that every modification of data is immediately sent to disk, which guarantees that nothing is lost in case of power failure.

[3] Waiting for many modifications before writing data to disk is worth because the transfer unit between memory and disk is a block (not a single byte at a time), and it is likely that many modifications of data fall in the same block in a short period of time.

[4] Consider the default block size is 8Kb.

[5] Since transactions are atomic, when they are cancelled because of one reason or another, all their changes must be completely undone, not leaving any trace at all.

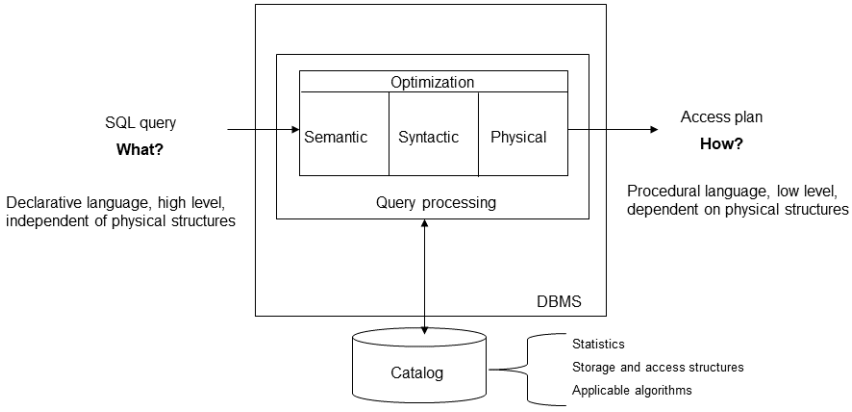


Figure 4.2: Query Optimizer

As sketched in Figure 4.2, the optimizer is a module inside the DBMS, whose input is an SQL query and its output is an access plan expressed in a given procedural language. Its objective is to obtain an execution algorithm as good as possible based on the contents of the database catalog:

- Contents statistics (for example, number of tuples, size of attributes, etc.).
- Available storage structures (for example, partitions and materialized views).
- Available access structures (for example, B-tree, bitmaps and hash indexes).
- Applicable algorithms (mainly for join and sorting).

The cost function to be minimized typically refers to machine resources such as disk space, disk input/output, buffer space, and CPU time. In current centralized systems where the database resides on disk storage, the emphasis is on minimizing the number of disk accesses.

The search space in this optimization problem is of exponential size with regard to the input query (more specifically NP-complex in the number of Relations involved), so not really all possibilities can be explored. Indeed, finding the optimum is so computationally hard, that it can result even in higher costs than just retrieving the data. Therefore, DBMSs use heuristics to prune the vast search space, which means that some times they do not obtain the optimum (altough they use to be close). Thus, in general, a DBMS does not find the optimal access plan, but only an approximation (in a reasonable time). It is important to know how the optimizer works to detect such deviations and correct them, whenever possible (for example, adding or removing some indexes, partitions, etc.).

Even though its real implementation could not be that modular, we can study the optimization process as if executed in three sequential steps (i.e., Semantic, Syntactic and Physical).

4.2.1 Semantic Optimization

This first module consists of transforming (i.e., rewriting) the SQL sentence into an equivalent one with a lower cost, by considering:

a) Integrity constraints

b) Logics properties (e.g., transitivity)

This module applies transformations to a given query and produces equivalent queries intended to be more efficient, for example, standardization of the query form, flattening out of nested queries, and the like. It aims to find incorrect queries (i.e., incorrect form or contradictory). Having an incorrect form means that there is a better way to write the query from the point of view of performance, while being contradictory means that its result is going to be the empty set. The transformations performed

depend only on the declarative, that is, static, characteristics of queries and do not take into account the actual query costs for the specific DBMS and database concerned.

Replicating clauses over one side of an equality to the other is a typical example of transformation performed at this phase of the optimization. This may look a bit naive, but it allows to use indexes over both attributes. For example, if the optimizer finds "a=b AND a=5", it will transform it into "a=b AND a=5 AND b=5", so that if there is and index over "b", it can also be used.

Another example of semantic optimization is removing disjunctions. For example, we may transform the disjunction of two equalities over the same attribute into an "IN". Reducing this way the number of clauses in the selection predicate would also reduce its evaluation cost. However, such transformation is not easily detected in complex predicates and can only be performed in the most simple cases.

In general, semantic optimization is really poor in most (if not all) DBMSs, and only useful in simple queries. Therefore, even though SQL is considered a declarative language, the way we write the sentence can hinder some optimizations and affect its performance. Consequently, a simple way for a user to optimize a query (without any change in the physical schema of the database) may be just rewriting it in a different way.

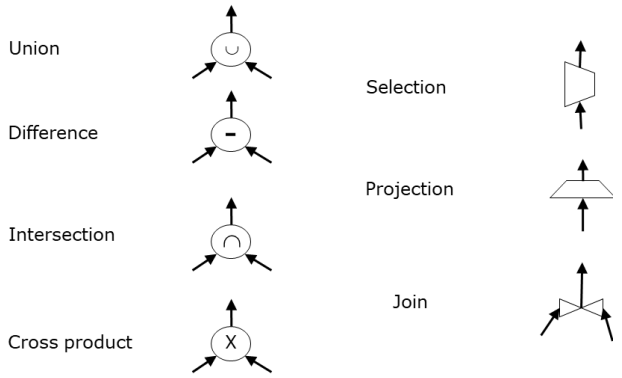


Figure 4.3: Graphical representation of Relational algebra operations

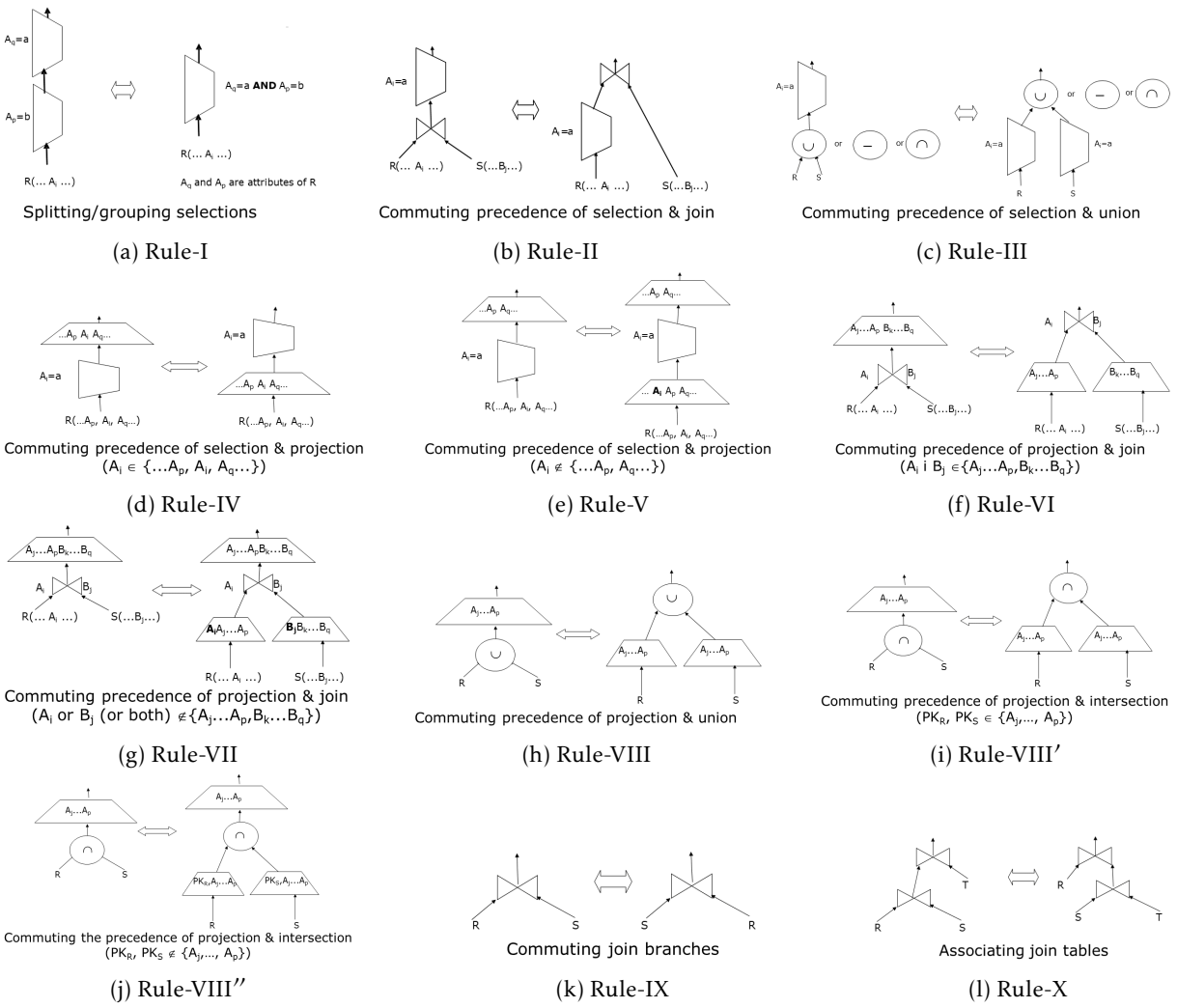
4.2.2 Syntactic Optimization

This second module consists of translating the sentence from SQL into a sequence of algebraic operations in the form of syntactic tree. There may be many sequences of algebraic operations corresponding to the same SQL query. Thus, the optimizer will choose the one with minimum cost, by means of heuristics. These sequences are usually represented in Relational algebra as formulas or in tree form (see Figure 4.3 for their graphical representation). The interpretation of the tree is as follows:

- Nodes
  - Root: Result
  - Internal: Algebraic Operations
  - Leaves: Relations
- Edges: Direct usage

This module determines the orderings of the necessary operators to be considered by the optimizer for each query sent to it. The objective is to reduce the size of data passing from the leaves to the root (i.e., the output of the operations corresponding to the intermediate nodes in the tree), which can be mainly done in two different ways: (i) reduce the number of attributes as soon as possible (i.e., reduce the width of the table), and (ii) reduce the number of tuples as soon as possible (i.e., reduce the length of the table). To do this, we use the following equivalence rules:

I. Splitting/grouping selections (see Figure 4.4a)



(a) Rule-I (b) Rule-II (c) Rule-III

(d) Rule-IV (e) Rule-V (f) Rule-VI

(g) Rule-VII (h) Rule-VIII (i) Rule-VIII'

(j) Rule-VIII'' (k) Rule-IX (l) Rule-X

Figure 4.4: Relational algebra equivalence/transformation rules

II. Commuting the precedence of selection and join (see Figure 4.4b)

III. Commuting the precedence of selection and set operations (i.e., Union, Intersection and Difference, see Figure 4.4c)

IV. Commuting the precedence of selection and projection, when the selection attribute is projected (see Figure 4.4d)

V. Commuting the precedence of selection and projection, when the section attribute is not projected (see Figure 4.4e)

VI. Commuting the precedence of projection and join, when the join attributes are projected (see Figure 4.4f)

VII. Commuting the precedence of projection and join, when some join attribute is not projected (see Figure 4.4g)

VIII. Commuting the precedence of projection and union (see Figure 4.4h). Notice that intersection and difference do not commute. For example, given $R[A,B] = \{[a,1]\}$ and $S[A,B] = \{[a,2]\}$, then $R[A] - S[A] = \emptyset$, but $(R-S)[A] = \{[a]\}$. Thus, similar to the case of join, to commute intersection/difference

with projection, we need to distinguish when the primary key of the tables is being projected (see Figure 4.4i) or not (see Figure 4.4j).

IX. Commuting join branches (see Figure 4.4k)

X. Associating join tables (see Figure 4.4l)

DBMSs use the equivalence rules to apply two heuristics that usually drive to the best access plan: (i) execute projections as soon as possible, and (ii) execute selections as soon as possible (notice that since they are heuristics, sometimes this does not result in the best cost). Thus, we will follow the algorithm:

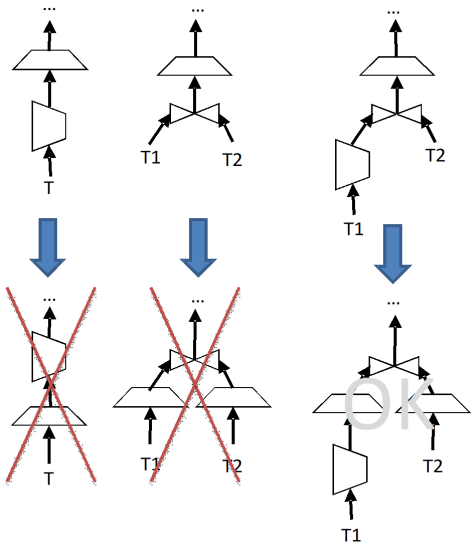


Figure 4.5: Rule exceptions

1. Split the selection predicates into simple clauses (usually, the predicate is firstly transformed into Conjunctive Normal Form – CNF).[6]
2. Lower selections in the tree as much as possible.
3. Group consecutive selections (simplify them if possible).
4. Lower projections in the tree as much as possible (do not leave them just on a table, except when one branch leaves the projection on the table and the other does not, see Figure 4.5).
5. Group consecutive projections (simplify them if possible).

It is important to notice that this algorithm only requires equivalence rules from I to VIII. The last two equivalence rules (i.e., IX and X) would be used to generate alternative trees. However, for the sake of understandability, we will assume this is not part of the syntactic optimization algorithm, but done during the next physical step.

It is also part of the syntactic optimization to simplify tautologies ($R \cap \emptyset = \emptyset$, $R - R = \emptyset$, $\emptyset - R = \emptyset$, $R \cap R = R$, $R \cup R = R$, $R \cup \emptyset = R$, $R - \emptyset = R$), and detect disconnected parts of the query, if any. Detecting disconnected tables (those with no join condition in the predicate) in a query can be easily done. However, in this case no error uses to be thrown. Instead, a cartesian product is performed by most (if not all) DBMSs. Moreover, in some cases, the tree is transformed into just a Directed Acyclic Graph (DAG) by fusing nodes if they correspond to exactly the same Relational operation with the same parameters (for example, the same selection operation in two different subqueries of the same SQL sentence, see Figure 4.6).

[6]Example of CNF: (x OR y) AND (z OR t OR . . .) AND...

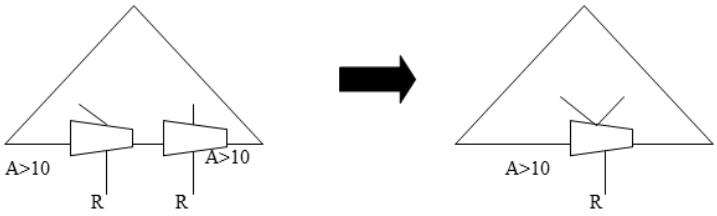


Figure 4.6: Non-tree syntactic tree

4.2.3 Physical Optimization

Given a syntactic tree, this module produces all corresponding complete access plans that specify the implementation of each algebraic operator and the use of any indices. Specifically, it consists of generating the execution plan of a query considering:

- Physical structures
- Access paths
- Algorithms

This is the main module of the query processor. It employs a *search strategy* that explores the space of access plans determined by the algebraic tree produced in the previous stage. It compares these plans based on estimates of their cost and selects the overall cheapest one to be used to generate the answer to the original query. First of all, to do it, we transform the syntactic tree into a process tree. This is the tree associated to the syntactic tree that models the execution strategy. Similar to the syntactic one, the interpretation of the process tree is as follows:

- Nodes
  - Root: Result
  - Internal: Intermediate temporary tables generated by a physical operation
  - Leaves: Tables (or Indexes)
- Edges: Direct usage

Notice that the main difference between both trees is that the new one is more expressive and concrete in the sense that it represents data, which can come from a user table, a temporary table, or even an index. Besides that, now the nodes represent real steps that will be executed (e.g., most projections disappear by fusing them with the previous operation in the data flow of just being removed if they lay on a table). Thus, to reduce the size of the search space that the optimization strategy must explore, DBMSs usually impose various restrictions. Typical examples include: never generating unnecessary intermediate results (i.e., intermediate selections and projections are processed on the fly). In a real optimizer, selections that are not on a table would also be fused with the previous operation, however we will assume they are not. Moreover, grouping and sorting operations are also added now to the process tree.

There are four steps in the physical optimizer:

1. Alternatives generation
2. Intermediate results cardinality and size estimation
3. Cost estimation for each algorithm and access path
4. Choose the best option and generate the access plan

During Step 1, we would firstly construct at this point all alternative algebraic trees by iterating on the number of Relations joined so far (using equivalence rules IX and X). The memory requirements and running time grow exponentially with query size (i.e., number of joins) in the worst case. Most queries seen in practice, however, involve less than ten joins, and the algorithm has proved to be very effective in such contexts. For a complicated query, the number of all orderings may be enormous.

Then, the physical optimizer determines the implementation choices that exist for the execution of each operator ordering specified by the algebraic tree. These choices are related to the available join methods for each join (e.g., nested loops, merge join, and hash join), if/when duplicates are eliminated, and other implementation characteristics of this sort, which are predetermined by the DBMS implementation. They are also related to the available indices for accessing each Relation, which are determined by the physical schema of each database.

During Step 2, this module estimates the sizes of the results of (sub)queries and the frequency distributions of values in attributes of these results, which are needed by the cost model. Some commercial DBMSs, however, base their estimation on assuming a uniform distribution (i.e., all attribute values having the same frequency). A more accurate estimation can be obtained by using a histogram, but this is a bit more expensive and complex to deal with.

During Step 3, this module specifies the arithmetic formulas used to estimate the cost of access plans. For every different join method, different index-type access, and in general for every distinct kind of step that can be found in an access plan, there is a formula that gives an (often approximate) cost for it. For example, a full table scan can be estimated as the number of blocks in the table B times the average disk access time D (i.e., Cost(FullTableScan) = $B \cdot D$)

Despite all the work that has been done on query optimization there are many questions for which we do not have complete answers, even for the most simple, single-query optimizations involving only Relational operators.[7] Moreover, several advanced query optimization issues are active topics of research. These include parallel, distributed, semantic, and aggregate query optimization, as well as optimization with materialized views, and expensive selection predicates.

4.3 Indexing

Indexes are data structures that allow to find some record without scanning the whole dataset. They are based on key-information pairs called entries. Usually, there is one of such entries for each record in the dataset, but not necessarily. The key is the value of one (or more) of the attributes in the record (typically an identifier), and the information can be a pointer, the record itself, etc.

The most used structures for entries are B-trees and hash-based. The former create an auxiliary hierarchical structure that can be traversed from the root, making decisions in each node depending on the keys, until entries are reached in the leaves. On the other hand, hash-based indexes divide the entries into buckets according to some hash function over the key. Then, on looking for some key value, we only need to apply the same hash function to that value to know in which bucket it is.

Usually, the indexed key corresponds to a single attribute, but nothing prevents us from using the data coming from more than one (i.e., concatenating several attributes in a given order to compose a somehow artificial key). Obviously, if we concatenate several values (e.g., in multidimensional fact tables the key can be easily composed of more than four attributes), the entries, and hence the corresponding index, will be larger (i.e., more expensive to maintain). For example, in case of building a B-tree structure, this will have more levels and will be more costly to traverse. Also, the structure will require updates more often, because now it involves more attributes that can potentially be changed. Nevertheless, it can still be worth, since searches are much more precise and using multiple attributes helps to discriminate better the records. This can also be done by using independent mono-attribute indexes, but these are more expensive to search.

The real problem with multi-attribute indexes is that the order of the indexed attributes is relevant. Indeed, to compare the value of attribute n, we need to fix the $n-1$ values of all the attributes concatenated before that. This would be specially limiting in the case of multidimensional queries, where

[7] If the query includes User Defined Functions, optimization is even harder.

the keys have many attributes and the user can define the search criteria on the fly (just one attribute missing in the user query can invalidate the index use).

4.3.1 Bitmap indexes

B-trees (also hash-based indexes) are quite popular, and specially useful for simple and very selective OLTP queries.[8] However, when we have more complex queries involving grouping, aggregation or many joins (like in OLAP systems), they do not result to be that useful because of the lack of selectivity and the many attributes involved. It is true that we can mitigate the problem by defining multiple multi-attribute indexes with different orderings (i.e., in general, as many as permutations of attributes in the key), but this would be really expensive to maintain.

At this point, it is important to realize that attributes in the key of a fact table do not have that many distinct values. It is only the composition of all dimensional identifiers that allows to distinguish every fact instance. Thus, if we index one of such attributes in the fact table, it will have many repetitions. When this happens, entries typically contain lists of pointers for the key value, which clearly reduces the size of the index since we have much less entries (only one for each distinct key value irrespectively of its repetitions).

We can now take this to the extreme and consider boolean attributes or with very few distinct values (e.g., age). In this case, our B-tree will have huge entries (with long lists of pointers inside), but not really any internal structure (just the root pointing to the leaves without any intermediate node, considering trees are n-ary for usually $n >> 100$). At this point, the structure of the entries (i.e., the tree) is not that important (mostly inexistent), but we rather need to consider how to efficiently encode the long lists of pointers (e.g., in a table with a million rows and a boolean attribute, each of its values would have associated half million pointers).

Ballpoint	Pencil	Pen	Rubber	A4 paper	A3 paper	Chalk	Eraser
1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0

Catalunya	León	Madrid	Andalucía
1	0	0	0
1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1
0	1	0	0
1	0	0	0
1	0	0	0

Figure 4.7: Examples of bitmaps

A list of pointers can be easily implemented as a fix-length vector (i.e., without extra cost or space for separators). Thus, given pointers of four bytes and a table of ten rows, we would require 40 bytes (i.e., 320 bits) to index it. Alternatively, we can encode the pointers as purely a bit structure, where each bit corresponds to the existence of a value in a row in the table, and we keep a list of bits for each different value in the domain of the attribute. Figure 4.7 exemplifies this for products (at LHS) and Spanish autonomous communities (at RHS). Thus, if we place all bits in a matrix, we have that columns represent domain values and rows correspond to rows in the table. A bit is set when the corresponding row has the value. A bit set in the first row of the first column means that the first row in the table has value "Ballpoint" for this attribute, and not any other value whose corresponding bits are not set.

Besides these matrixes being much smaller that the list of pointers (i.e., 80 bits for LHS and only 40 for RHS vs 320 bits for the list of ten pointers of four bytes each), they are also very easy to maintain by just switching bits. Also, evaluating over bitmap indexes complex predicates with conjunctions, disjunctions and negations is equivalent to doing the same operations at bit level, which is much easier and efficient than operating lists of pointers.

[8] RDBMSs automatically create a B-tree associated to every primary key declared.

From the point of view of query optimization, it is important to remember that we need to account for the number of blocks being accessed. Therefore, being R the number of records in a block and SF the selectivity factor of the query predicate, we can easily estimate the probability of accessing a block like $1-(1-SF)^R$. This probability gives us the expected number of table blocks being accessed. To the number of table blocks, we need to add the blocks required to store the bitmap, which as already said is expected to be much smaller than a B-Tree, but depends on the number of different values in the predicate (the more values, the more lists of bits we need to retrieve).

In general, we can say that bitmap indexes are better than B-trees for multi-value queries with complex predicates. Indeed, they result in optimum performance for predicates with several conditions over different attributes (each with a low selectivity), when the selectivity factor of the overall predicate is less than 1%. Bitmap index is sometimes assumed to be useful only for point queries, but in some implementations (e.g., Oracle), bitmaps can be used even for range queries. Without compression, bitmap indexes would use more space than lists of pointers for domains of 32 values or more (assuming pointers of four bytes). However, we can easily assume that they require less space, due to compression. It is because of this that they degrade performance in the presence of concurrent modifications of data, since they require to lock too many granules for every individual update (all granules compressed in the same block are locked at once).

4.3.1.1 Compression

Assuming mono-valued attributes, notice that every row in each matrix can only have one bit set. Consequently, in the whole matrix, we have as many attributes set as rows, and the matrix results to be really sparse (the more values, the more sparse the matrix is).[9] It is well know that the more sparse a matrix, the more compressable it is.

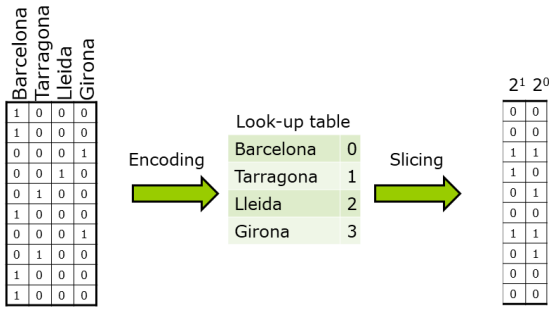


Figure 4.8: Examples of bit-sliced compression

The first technique to encode the bit matrix is known as "bit-sliced". It simply assigns an integer value to each domain value (e.g., Barcelona could be 0, Tarragona could be 1, Lleida could be 2 and Girona could be 3), and keep this assignment in a lookup table. Then, we assign the corresponding integers to each row depending on its value. The final matrix corresponding to the bitmap index would be the binary encoding of those integers, as exemplified in Figure 4.8.

Alternatively to bit-sliced, we can also use run-length encoding. In this case, we define a *run* as a sequence of zeros with a one at the end (i.e., $0\cdots01$). Then, we need two numbers to encode the run, where the first number (i.e., content length) determines the size of the second (i.e., true content, which corresponds to the run length). The important thing is that content length and content (a.k.a. run length) need to use a different coding mechanism to be able to distinguish them.[10] Indeed, the content is simply a binary encoding of the run length (e.g., for a run of length five, we would encode it like 101), and the length of the content is imply a list of ones ending in a zero (e.g., for the previous number of

[9] As a rule of thumb, we should define bitmap indexes over non-unique attributes whose distinct values have hundreds of repetitions (i.e., $ndist < \frac{|T|}{100}$).

[10] Notice that using two codes is mandatory for variable length coding, because without knowing the beginning and end of every number we could not disambiguate the coded bits (i.e., we need some separating mark, which in this case is the length of the next coded number).

length three, we would encode such length like 110). Thus, after the encoding, a run 000001 of length five would be encoded like 110101.

At this moment, it is important to realize that we can operate encoded bit-vectors without decompressing them. Indeed, to perform a disjunction (a.k.a. OR), all we need to do is traverse in parallel both encoded bit-vectors generating in the output a row at the end of a run of either of them. Being $l_A(i)$ (respectively $l_B(i)$) the length plus one (i.e., overall number of zeros in the run plus one) of the i^{th} run of attribute A (respectively attribute B):

$$\text{A OR B} \rightarrow \{a \mid \exists x, a = \sum_{i=1}^{x} l_A(i)\} \cup \{a \mid \exists x, a = \sum_{i=1}^{x} l_B(i)\}$$

For the conjunction (a.k.a. AND), we also traverse in parallel both encoded bit-vectors but now we generate a row in the output when the end of a run coincides in both of them. Therefore:

$$\text{A AND B} \rightarrow \{a \mid \exists x \exists y, a = \sum_{i=1}^{x} l_A(i) = \sum_{i=1}^{y} l_B(i)\}$$

4.3.1.2 Indirection

At this moment, it is important to point out that the bitmaps per se just give me the relative position in the table of the desired rows. If rows would be of fix length, we could get the physical position by just multiplying the offset by the row size. However, rows can actually be of variable length. Therefore, if this is the case, we need some mechanism that translates row numbers into physical positions in the disk.

The first simple possibility is creating an auxiliary index with the row number as key and the physical position as information in the entries. Although this is feasible and would still provide some benefits over pure B-tree indexes, there is an alternative to avoid it. All we have to do is to determine and fix a maximum number of rows in each disk block (usually known as Hakan factor). Thus, we will assume each block has such maximum number of rows and we will index them as if they exist. For example, if we decide (based on statistical information of the table) that each block can have a maximum of five rows, but the first one has only four, we would index five anyway (kind of assuming the existence of a phantom in the block). Obviously, this inexistent row will not have any value because it does not really exist, so it will only generate more zeros in the bitmap index and hence longer runs (but never more runs). Consequently, if we properly adjust the maximum number of rows per block, the effect of assuming the existence of this fictitious rows is minimal in the size of the compressed bitmap index.

4.4 Join

Join is the most expensive operation we can find in process trees (together with sorting). That is why we find many different algorithms implementing it: Row Nested Loops (a.k.a. Index Join), Block Nested Loops, Hash Join, Merge Join, etc. We are going to pay attention now to the first two, because of being the most basic ones.[11]

Row Nested Loops consists on simply scanning one of the tables (i.e., outer loop), and then try to find matches row by row in the other table. In its simplest (also useless) version, it scans the second table in an inner loop. It is obvious that scanning the second table for every row in the outer loop is the worst option. Thus, an improvement of the algorithm is using an index (if this exists) instead of the inner loop. This option is specially worth when the outer loop has few rows, so we only need to go through the index few times.

At this point, it is important to remember that the disk access unit is the block and not the row. Thus, we do not bring rows one by one in the outer loop, but we bring R rows at once. Therefore, an alternative improvement to the algorithm, known as Block Nested Loops, is to indeed scan the second table in an inner loop but only once per block (not once per row). Since the whole block is in memory,

[11] From here on, we assume they are binary operators, but there are also multi-way join algorithms available in many DBMSs.

we can try to match all its rows in a single scan. Actually, we can bring into memory as many blocks as they fit (not necessarily only one). Taking this to the extreme, we might bring all the external table into memory and then scan the second one only once. To achieve this and making use of the commutative property of join, we will always take the smallest table for the outer loop.

4.4.1 Advanced join techniques

Since joins are the most expensive operations, many DBMSs try to either detect some patterns to apply specific techniques (which usually gives better results than generic ones), or implement specific indexing structures.

```
SELECT ...
FROM Sales f
WHERE f.prodId IN (SELECT d1.ID FROM Product d1 WHERE d1.articleName IN ("Ballpoint","Rubber"))
AND f.placeId IN (SELECT d2.ID FROM Place d2 WHERE d2.region="Catalunya")
AND h.timeId IN (SELECT d3.ID FROM Time d3 WHERE d3.month IN ("January02","February02"))
GROUP BY ...
```

Figure 4.9: Star-join pattern

4.4.1.1 Star-join

Multidimensional queries involve the join of a fact table against all its many dimension tables. However, this can be hidden in the query, so that the pattern is not obvious. Thus, some semantic optimizers try to transform the user query so that the main query corresponds to the fact table, and the access to the dimension tables is encapsulated in independent subqueries. On finding this new shape (exemplified in Figure 4.9, the physical optimizer will access the dimensions first and try to extract their IDs to later use them to go through indexes (potentially bitmaps) over the primary key of the fact table. The results of the subqueries can be even temporarily kept in memory to avoid retrieving them more than once from disk, if this would be necessary.

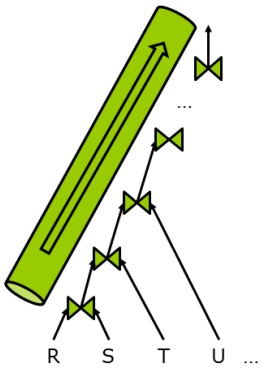


Figure 4.10: Pipelining sketch

4.4.1.2 Pipelining

When tables are really large (like in the case of fact tables), they do not fit in memory. Thus, the result of every operation in the process tree should be stored in the disk, so that the next operator can take over the processing. Clearly, writing and reading each intermediate result generates an extra cost, that we should try to avoid. Indeed, some join algorithms allow to process one row at a time. Thus, every row can be pipelined in memory through a series of joins as sketched in Figure 4.10. This way, first row of R will be joined against S, then the result against T and so on and so forth until the end of the

process tree. When this first row has gone through all the joins, we can start processing the next row and continue like that until we finish the whole table R. In doing it this way, we only need to have one row of R in memory and avoid materialization of intermediate results. This does not work with all join algorithms, but it does with Row/Block Nested loops. It is also important to notice that for this to really work to full potential, the process tree must be left-deep like the one in the figure. In this case, the cost of the query is given by the different access paths of the tables (i.e., that of their indexes if we consider using Row Nested Loops).

4.4.1.3 Join-index

On talking about indexes containing entries of key-information pairs, we assumed that both the key and the information correspond to a single table. Nevertheless, nothing prevents us from using keys in one table (e.g., a dimension) to index data in another one (e.g., a fact table). Thus, we can use a single index to enter through the dimensional values and then find the rows in the fact table (i.e., somehow join them). For example, we could index sales by the region where it took place. The key of the index would be the attribute `region` of the location dimension and the information would be a list of pointers (or bitmap) to the corresponding sales in the fact table.

Multimedia Materials

Bitmap Indexes (en) 

Bitmap Compression (en) 

Join Algorithms (en) 

Advanced Join Techniques (en) 

Chapter 5

Materialized Views

In the 60's, before the Relational model, several alternatives co-existed to store data (e.g., hierarchical, network, etc.). In all these cases, the user had to know the structure of files to be able to access the data, and any change in those files had to be reflected in the applications using them. Thus, both applications and data files were tightly coupled. Then, the big achievement of the Relational model was to provide a higher level of abstraction, that made data management independent of how the data were physically stored.

Figure 5.1: 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 5.1, defined three different levels for DBMSs to implement. At the RHS, we had the internal one corresponding to files and data structures like indexes, partitions, etc. To its left laid the tables 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, because it made a difference with regard to predecessors. In this way, views are like windows to a database that provide access to only a portion of the data that is either of interest or related to an application, and at the same time allow to automatically reshape those data according to user needs. However, applications do not make any distinction between tables and views, since they can be queried in exactly the same way.

Nevertheless, internally, a table has a schema (i.e., name and attributes), while a view has a schema, but also a query that defines its content with regard to that of some tables (this is why, views are sometimes called "derived relations" or "named queries"). Thus, views existence is linked to that of the corresponding tables, sometimes referred to as "base tables" (e.g., you cannot drop a table that has views defined over it).

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

5.1 Materialized Views

The query defining a view is stored in the catalog of the database, and every time a user accesses the view, that query must be executed. Clearly, this incurs in an extra cost for views that tables do not have. A way to solve it is running the query only once and store its results. Now, every time a user accesses the view, we do not need to re-run the query, but just retrieve the stored results. This is called a "Materialized view" (MV).

From the DW viewpoint, MVs open a new door to optimization. A well known way to optimize a query is building indexes (both primary and secondary), as explained in Chapter 4. Now, complementarily, we can also precompute results of expected queries in the hope that such results will be smaller than the base tables. Indeed, users usually retrieve a subset of the attributes, some tuples, or perform some grouping. Consequently, query results use to be smaller than the data underneath (i.e., the MV will be smaller than the base table), but it is not only the saving in the amount of data that has to be accessed, but also the saving in the execution time of the operations we avoid (e.g., groupings, joins). In the case of a DW, given the size of fact tables, such savings are specially important and often more relevant that those we can obtain using indexes (since selectivity factors are not small enough for B-trees to be useful in multidimensional queries).

Obviously, storing the results of the query is convenient, but also redundant (in the sense of being not really necessary). As any kind of redundancy, it requires extra disk space, but also maintenance. Indeed, as soon as the base tables do not change, everything is fine. However, if the base tables change, the changes must be reflected in the MVs or otherwise query results would be inconsistent (i.e., querying a view would give results different from those obtained by querying its base tables). Therefore, views (materialized or not) generate some problems the DBMS has to deal with.

5.2 Problems associated to views

Using views simplifies the management of the database and promotes its usage (since it approaches data to the way users see them). Moreover, on top of that, if we materialize the content of (some) views we also affect performance. Nevertheless, the relationship between views and the corresponding base tables generates different problems:

1. View expansion (i.e., transform a query over views into a query over base tables)
2. Answering queries using views (i.e., transform a query over base tables into one over MV)
3. View updating or View maintenance (i.e., propagate changes in the base tables to the corresponding MVs)
4. Update through views (i.e., propagate changes expressed over views to the corresponding base tables)

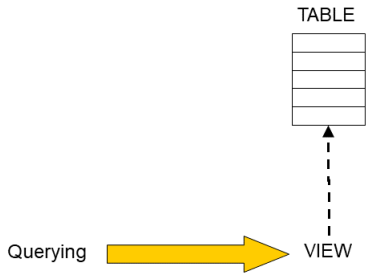


Figure 5.2: View Expansion

5.2.1 View Expansion

This is the most basic and easy to solve problem related to views. Figure 5.2 sketches it, and represents that a query over the views has to be transformed into a query over the base tables to actually retrieve the data. This affects only non-materialized views, because their data is actually not stored anywhere and needs to be recomputed every time the user poses a query. Notice that if the view is materialized, the query is simply answered based on the stored data.

The solution to the problem is as simple as replacing the view name in the user query by the corresponding view definition stored in the catalog.

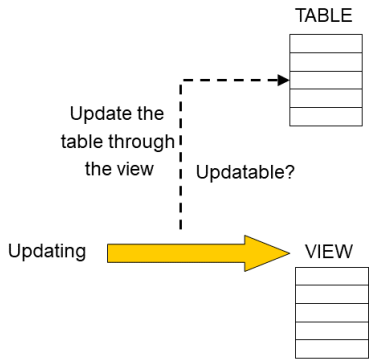


Figure 5.3: Update Through Views

5.2.2 Update Through Views

Since a view is just a Relation, we should be allowed to operate it as if it were a table. However, it does not really have its own data, but they are given by the corresponding base tables. Consequently, we need to propagate the changes expressed in terms of views to the base tables by means of a translation process. Figure 5.3 sketches the problem we find on receiving a statement updating a view, that then has to be actually executed in the tables. This is explained in [Vel09] like "Since the view instance depends on the instances of the base tables, to execute an update on the view one needs to find a number of base table modifications whose effect on the view instance is the modification described by the update command". It is important to realize that this is independent of whether the view is materialized or not, because anyway we have to keep the database consistent.

The problem is not trivial, in general, and it is even unsolvable if the view definition contains aggregates or joins, because in these cases there is potentially more than one change in the tables that correspond to a single change in the view. Therefore, the real question is whether a view is updatable or not.

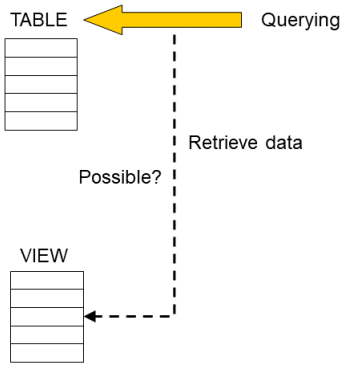


Figure 5.4: Answering Queries Using Views

5.2.3 Answering Queries Using Views

In case of materializing a view, this might be much smaller than the original tables (having less tuples and/or attributes). Thus, it could be more efficient to answer queries over the materialized result instead of using the base tables (actually, this is the purpose of materializing a view). Nevertheless, we do not want (or expect) the user to be aware of query costs or even the existence of MVs. Therefore, the system should detect automatically when a query expressed over the base tables can be answered more efficiently using some already existing MVs. If this is the case, it should also transform an arbitrary query over the tables into a query over the corresponding MVs.

Figure 5.4 sketches the problem of detecting if it is possible to use the data in an MV to answer a query expressed on a table. As defined in [Vas09], the problem refers to "... given a query Q over a database schema Σ, expressed in a query language L_Q and a set of views $\{V_1, V_2, \ldots, V_n\}$ over the same schema, expressed in a query language L_V, is it possible to answer the query Q using (only) the views $V_1, V_2, \ldots, V_n$?".

Basically, to know if it is possible to rewrite a query in terms of existing MVs we need to check if:

a) The query predicate is subsumed by that of the view (i.e., the where clause in the query logically entails the where clause in the view).

b) The aggregation level (a.k.a. group by) in the query must be coarser or equal to that in the MV (functional dependencies can be used to check it).

c) Aggregates (i.e., sum, min, etc.) must coincide (or be computable somehow from the information in the view).

Checking these conditions (specially (a)) is already known as a computationally complex task (see [Hal01]). Consequently, DBMSs offer the possibility of manually enabling or disabling the participation of each MV in answering queries over tables (the less views enabled, the cheaper the checking is). Anyway, DBMSs never perform an exhaustive search, but just apply some promising, pre-defined rules.

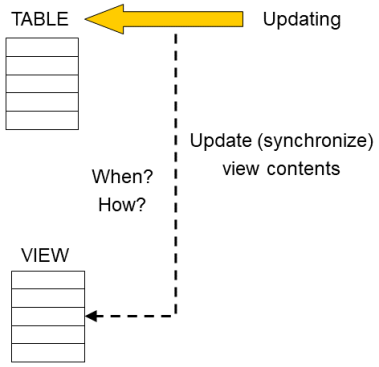


Figure 5.5: View Updating

5.2.4 View Updating

As already mentioned, the cost of materializing views is not that much in the extra space they require, but rather on keep them synchronized with the base tables. Indeed, every change in the base tables must be potentially propagated to the MVs, as sketched in Figure 5.5. Thus, according to [LS09], view updating (or maintenance) typically refers to "the updating of a materialized view to make it consistent with the base Relations it is derived from".

In general, we would say that tables and materialized views must be consistent at all time. However, in some cases, we can allow some temporal inconsistency (this clearly depends on the application using the data) and wait until we have a batch of changes to apply all of them at once (i.e., deferred) for the sake of efficiency.

Moreover, we should not only think about when to propagate, but also which is the best way to do it. It is obvious that we can always take the view definition from the catalog and re-execute it to supersede the whole current (outdated) view content. Nevertheless, it should also be obvious that this is not worth when only one single row has changed and you can modify it individually leaving the rest unaffected. The former is know as "complete" update, while the latter is called "incremental". Incremental update is always possible (even deferred), as soon as you keep track (in a separated file/table/log) of all the required information to do it. Therefore, the problem is to decide which option is the best. When all tuples have changed, it is better to perform a complete update, however, when very few changed, an incremental one is more efficient. The problem is then to decided in the general case when to use one or another depending on the estimated number to tuples affected by changes.

5.3 Materialized View Selection

As we have seen, there are alternatives to fine tune the materialized view maintenance. However, this is never for free. Therefore, we much carefully choose which MVs are worth the maintenance cost and which are not. The less a view changes and the more it is queried the better to materialize it. Oppositely, views that change often and are rarely queried are clearly not good candidates to be materialized. We just need to analyze the trade-off between the overhead of maintaining a view and the saving it provides in queries.

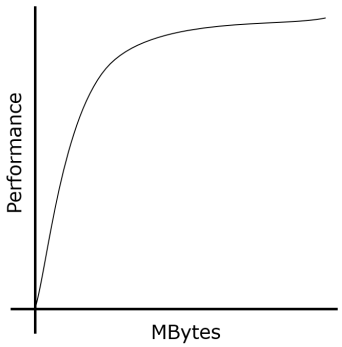


Figure 5.6: Materialization Trade-Off

Resources are always finite, thus, even if there are many queries that potentially improve performance, often it is not possible to materialize them all. It could be simply a lack of disk space, but usually it is rather a lack of time to keep all of them up to date. Just realize that resources (a.k.a. time) we can devote to maintain MVs are limited.[3] Figure 5.6 depicts the relationship between performance gain and space use in view materialization (we could draw the same for update time duration instead of MBytes in the horizontal axis). We should interpret this in the sense that by devoting few MBytes to view materialization, we can improve performance a lot. However, spending more and more space (also time) is only going to result in marginal performance improvement. This does not mean that materializing any view magically improves performance. We need to wisely find the ones maximizing the impact.

Only considering the usage of a group by clause, the number of candidate views to be materialized is exponential in the number of attributes of the table (this is just worsened by considering also where clauses in the queries). Consequently, an exhaustive search is simply impossible. Instead, we should use heuristics and greedy algorithms to choose MVs (see [GR09]). For example, given a set of queries whose performance we want to optimize, we should only consider views that have exactly the same group by clause as some of those queries, or the union of some of them.

[3]The available time to maintain MVs is usually called "update window", and inside it, users are banned from querying.

Multimedia Materials

Materialized View Problems (en) 

Answering Queries Using Views (en) 

View Updating (en) 

Materialized View Selection (en) 

Chapter 6

Extraction, Transformation and Load

There are many situations in which we have to move data from one database to another, which implies Extracting the data, then potentially performing some Transformation to finally Load them in the destination. Some situations where we can find this are:

- Restructure data to be used by other tools
- Transactional data safekeeping
- Use multiple sources together
- Improve data quality by removing mistakes and complete missing data
- Provide measures of confidence in data (e.g., filtering out erroneous records)

This kind of data intensive flows is so common that has created the need of specialised tools like Microsoft Data Transformation Services[1], Informatica Power Center[2], Pentaho Data Integration[3], and many others.

6.1 Preliminary legal and ethical considerations

Before creating any data migration flow, we should ask ourselves if we are really allowed to do it, and even if we are, whether this is ethically acceptable. That the technology allows to do something, does not mean that it should be done. In the end, it is the people (i.e., data engineers in this case) not the tools alone that perform the tasks. Thus, this job comes with a big responsibility[4].

This is specially true when we are talking about personal data. Due to that, there is a specific EU directive to regulate what can and cannot be done, as well as different agencies to safeguard the rights of protection of personal data at different administrative levels, like the Catalan one[5].

Thus, before doing anything, you should ask yourself:

1. Who is the owner of the data?
2. Are we allowed to process them?
3. Are we allowed to use them for this purpose?
4. Will we keep confidentiality of data?
5. Did we implement the required anonymization and inference control mechanisms?
6. How will the results of my work be used?

[1] https://en.wikipedia.org/wiki/Data_Transformation_Services
[2] https://www.informatica.com/gb
[3] https://www.hitachivantara.com/en-us/products/data-management-analytics/pentaho-platform.html
[4] http://wp.sigmod.org/?p=1900
[5] https://apdcat.gencat.cat/en/inici/index.html

6.2 Definition

As we said, we can find ETL flows in many situations and systems, but we should focus now on DW. If we pay attention to the different architectures in Chapter 2, it is easy to see that the ETL layer is explicit in case of two- and three-layers. However, it is also implicit in some other places including the single-layer architecture. In Fig. 2.2, we see it between the sources and the DW, but there is also data movement between the DW and the DMs. In Fig. 2.3, it is again explicit between the sources and the Reconciled data, but there is also data movement from this to the DW and then to the DM. Finally, it is not that obvious, but some data movement is implicit in the virtual DW layer of Fig. 2.1, too.

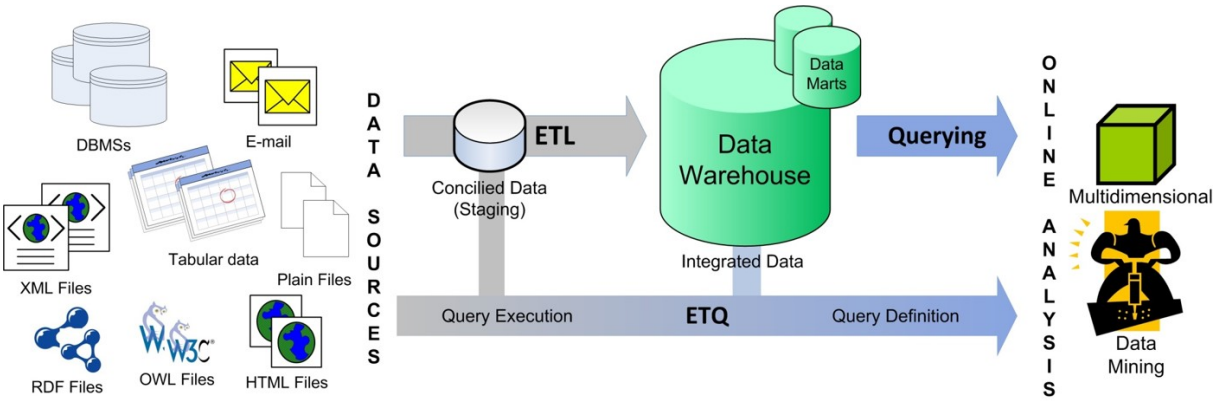


Figure 6.1: ETL flows

These different possibilities are generalized in Fig. 6.1, where we distinguish between ETL (corresponding to two- and three-layers) and ETQ where the data are not loaded anywhere, but directly Queried by the final user without any materialization (corresponding somehow to single-layer architecture). Even more generally, we can also find some authors and software providers proposing and advocating for an ELT variant, where extracted data are firstly loaded into a powerful DBMS to then transform them inside using ad-hoc SQL statements.

6.2.1 Extraction

Some data tasks require multiple and heterogeneous sources, that can be in different supports (e.g., hard disk, Cloud), different formats (e.g., Relational, JSON, XML, CSV), from different origins (e.g., transactional systems, social networks), and with different gathering mechanisms (e.g., digital, manual). So, it will be the first task of the ETL to solve all these differences as data moves through the pipe.

Besides purely formatting, since our DW must be historic, it is specially important to pay attention too to the differences in the temporal characteristics of sources:

Transient: The source is simply non-temporal, so if we do not poll it frequently, we'll miss the changes (any change between two extractions will be lost).

Semi-periodic: The source keeps a limited number of historical values (typically the current and the previous one), which gives us some margin to space extractions, so reducing the disturbance to the source.

Temporal: The source keeps an unlimited number of historical values (but may still be purged periodically), which gives complete freedom to decide the frequency of extraction.

Fig. 6.2 sketches the different mechanisms we have to extract data from a DBMS:

a) Application-assisted: Implies that we modify the application and intercept any change of the data before it goes to the DBMS.

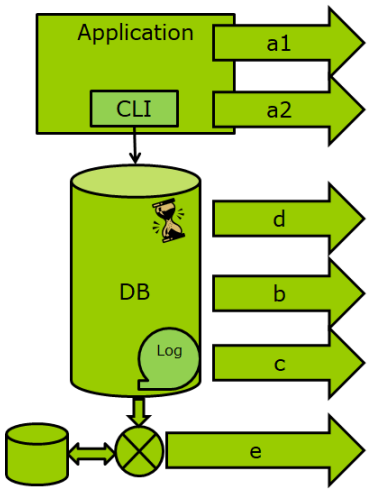


Figure 6.2: Data extraction mechanisms

1) This can obviously be done by modifying the code (e.g., PhP), and injecting the required instruction to push any change in the data to the ETL pipe of the DW. The obvious problem of this approach is that it will be expensive to maintain (any application accessing the DBMS must be modified) and hardly sustainable.
2) A more elegant alternative is to modify the Call Level Interface (i.e., JDBC or ODBC driver) so that every time a modification call is executed, this is replicated in the ETL pipe of the DW. Obviously, this is more sustainable than the previous, but can be harder to implement and some times expensive to execute (interactions to the DBMS are more expensive and this would still impact the application performance).

b) Trigger-based: If the DBMS provides triggers, we can use them to intercept any change in the database and propagate it to the DW. The limitation is obviously that not all DBMS provides triggers, and they may also result too expensive.

c) Log-based: Any DBMS provides durability mechanisms in the form of either logs or incremental backups. This can easily be used to efficiently extract the data. The main problem is that depending on the software provider, it may be using proprietary mechanisms and formats, which are hard to deal with.

d) Timestamp-based: Some databases are temporal (or even bi-temporal) and already attach timestamps to any stored value (or can be easily modified to do so). If allowed and does not generate much overhead, this would be the best option, because offers the maximum flexibility to the extraction. However, it requires having the control over the source and the power to impose the modification.

e) File comparison: A simple option is to perform periodic bulk extractions of the database (a.k.a. backups). The first time, we would push all of it to the DW and keep a copy in some staging area. From there on, every new extraction would compare against the previous one in the staging area to detect the changes and only push those to the DW.

Besides these technical issues, in case of existing alternatives, we should also consider the interest and the quality of the sources in the choice. Firstly, a source should always be relevant to the decision making (i.e., it should provide some data to either a dimension or fact table). In general, having more data does not mean making better decisions (it can be simply harder or even confusing if data are irrelevant or wrong)[6]. On the other hand, it should not be redundant, because it would simply be a waste of resources. If there are some alternative sources for the same data, we should chose the one

[6] https://dangerousminds.net/comments/spurious_correlations_between_nicolas_cage_movies_and_swimming_pool

of highest quality in terms of completeness, accuracy, consistency and timeliness. A good reason to extract the same data from more than one source would be that we can obtain higher quality from them together.

6.2.2 Transformation

The second part of the ETL consists of standardizing data representation and eliminating errors in data. It can be seen in terms of the following activities, not necessarily in this order (actually, due to its difficulty, its development is an iterative process estimated to take 80% of the knowledge discovery process):

Selection has the objective of having the same analytical power (either describing or predicting our indicators), but with much less data. Obviously, we can decide not to use some source, but even on taking one source, we can use it partially by reducing:

a) Length (i.e., remove tuples) by sampling (might be interesting not to remove outliers), aggregating (using predefined hierarchies) or finding a representative for sets of rows (i.e., cluster).

b) Width (i.e., remove attributes) by eliminating those that are correlated, performing an analysis of significance, or studying the information gain (w.r.t. a classification).

Integration has the purpose of crossing independent data sources, which potentially have semantic and syntactic heterogeneities. The former requires reshaping the schema of data, while the latter is solved tranforming their character set and format.

Cleaning includes generating data profiles, splitting the data in some columns and standardize values (e.g., people or street names). However, the main purpose is to improve data quality in terms of:

- Completeness by imputing a default value (or even manually assigning different ones if not many are missing), which can be either a constant, some coming from a complementary source/lookup table, the average/median/mode of all the existing ones, the average/median/mode of the corresponding class, or the one maximizing the information gain.
- Accuracy by detecting outliers performing some variance analysis, evaluating the distance to a regression function, or identifying instances far from any cluster.
- Correctness, by checking constraints and business rules, or matching dictionaries/lookup tables.

Feature engineering derives new characteristics of the data that are expected to have more predictive power.

Preparation sets the data ready for a given algorithm or tool. In some cases, this requires to transform categorical attributes into numerical ones (just creating some encoding), but others it is the other way round and numerical attributes need to be discretized (e.g., by intervals of the same size, intervals of the same probability, clustering, or analysis of entropy). Some algorithms are also affected by differences in scales, so numerical attributes need to be normalized (e.g., dividing by the maximum $\frac{x}{max}$, dividing by the domain size $\frac{|x-min|}{|max-min|}$, dividing by the standard deviation $\frac{x-\mu}{\sigma}$, or simply dividing by some power of ten $\frac{x}{10^j}$). Some other algorithms require the transformation of data into metadata by pivoting and converting rows into columns.

6.2.3 Load

The last phase is loading the data in the target of the flow[7]. A typical technique to facilitate this in a DW environment is using the concept of *update window*. This is a period of time (typically at night or non-working days) during which analysist cannot access the DW (so they cannot interfere in the loading). Separating user queries from ETL insertions, we firstly save the overhead of concurrency

[7] As previously said, we could also send it directly to the user ready to be consumed in an ETQ

control mechanisms, which can be a significant gain by itself, but also allows to disable materialized view updates and indexes (since users are not using them). Once the load is over, we can rebuild all indexes and update all the materialized views in batch, which is much more efficient than doing it incrementally and intertwined with insertions.

6.3 ETL Process Design

Many organizations are opting for an ad-hoc, hand-coded ETL processes, mostly due to no established standards and tools for conceptual modeling of the ETLs. However, such settings soon become unmanageable with hundreds of scripts with usually poor documentation and little or no reusability of the code.

Conceptual ETL process design: Many research attempts propose conceptual modeling for ETL to provide better understandability and sharing of ETL processes as well as their maintenance and reusability. Some are proposing either novel, ad-hoc notation [SV03], or based on well-known modeling languages, like UML [TL03] or BPMN [VZ14]. Besides better documentation of the ETL processes in an organization, such approaches using a strictly defined conceptual model for designing an ETL process allow for automation of code generation. One of the main tasks of the ETL process conceptual design is the identification of schema mappings between data source schemata and target DW schema [SV18]. Furthermore, the conceptual design also specify a primary data flow that describes the route of data from the sources toward the data warehouse, as they pass through the transformations of the workflow.

Logical ETL process design: At the logical level, we must specify the detailed workflow for executing our ETL process. On the one hand, we specify a **data flow** in terms of what transformation each ETL operation does over the data. On the other hand, we specify a **control flow**, which takes care of scheduling, executing, and monitoring of the ETL process. In addition, we can as well specify recovery plans in case of failure occurrences. Most ETL tools provide the designer with user friendly (drag&drop) functionalities to construct both the data and the control flow for an ETL process (e.g., Pentaho Data Integration - Kettle).

Physical ETL process design: Finally, we need to provide an executable version of the previously designed ETL process. While ETL tools conveniently integrate logical ETL design and its execution, hand-coded solutions require the implementation of procedures and script that fulfill the data flow and control flow tasks.

- **Hand-coded ETL.** As previously discussed, to implement an ETL, we do not really need an ETL tool. It can be done programmatically with a skillful programming team, as well. The advantage of this is that we are not limited by the features of the tool, we can reuse legacy routines as well as know-how already available. A good example of this approach lately is MapReduce, which facilitates processing schemaless raw data in read-once datasets, cooking them before being loaded in a DBMS for further processing. However, in general, such programmatical approach only works for relatively small projects. If the project has a certain volume and requires some sophisticated processing, manual encoding of data transformations is not a good idea, specially from the point of view of the maintenance and sustainability in the long term.

- **ETL tools.** Quoting Pentaho, "The goal of a valuable tool is not to make trivial problems mundane, but to make impossible problems possible". An ETL tool, like Pentaho Data Integration, cloverETL, JasperETL, or Talend, firstly offers a GUI that facilitates the encoding of the flows. Moreover, they provide some metadata management functionalities and allow to easily handle complex data type conversions as well as complex dependencies, exceptions, data lineage and dependency analysis. Also, they facilitate common cumbersome tasks like scheduling processes, failure recovery and restart, and quality handling.

6.4 ETL Process Quality

Being complex and time-critical components inside the Business Intelligence systems, ETL processes require careful consideration when it comes to fulfilling different process quality characteristics. In

addition, unlike other business processes, ETL process has an important quality dimension related to the quality of data. You can find more details about **data quality** in Chapter 7, while we dedicate this section to **ETL process quality** dimensions. In particular, we emphasize the following important **ETL process quality** dimensions, while more extensive list can be found in [The17].

- **Performance.** Time behavior and resource efficiency are the main aspects that have traditionally been examined as optimization objectives for data processing tasks (e.g., query optimization). Currently, there is almost no, or very limited automation for the ETL process optimization in ETL tools, and it is mainly relying on the support provided by the data source engines (e.g., RDBMS, Apache Pig) and for part of ETL process flow (e.g., pushing Relational algebra operators to the data source engine whenever possible). In addition, ETL tools, like Pentaho Data Integration allows manual tuning of the ETL process, by configuring the resource allocation (memory heap size) or parallelizing ETL operations, a solution limited by the hardware capacity of the host machine. However, many research attempts have been exploring the topic of optimizing the complete ETL process, building upon the fundamental theory of query optimization [SVS05]. The main limitation is still on the complex ("black-box") ETL operations (those not being able to be expressed using Relational algebra).

- **Reliability.** Reliability of ETL processes represents the probability that an ETL process will perform its intended operation during a specified time period under given conditions. At the same time, in the presence of a failure, the process should either resume accordingly (*recoverability*) or should be immune to the error occurred (*robustness* and *fault tolerance*) [SWCD09]. Fault-tolerance for example can be improved either by *replicating* the flow (i.e., running in parallel multiple identical instances of a flow) or flow *redundancy* (i.e., providing multiple identical instances of a flow and switching to one of the remaining instances in case of a failure). For *recoverability*, the most usual technique is to introduce recovery check (i.e., persisting data on the disk after heavy operations, such that in the case of failure we can restart the ETL process execution from the later and partially reusing already processed data).

- **Auditability.** Auditability represents the ability of the ETL process to provide data and business rule transparency [The17]. This includes *testability*, or the degree to which the process can be tested for feasibility, functional correctness and performance prediction; and *traceability* which includes the ability to trace the history of the ETL process execution steps and the quality of documented information about runtime. Well-documented ETLs (e.g., by means of providing conceptual and logical design) enable better auditability of the entire process and easier testing of its specified functionalities.

- **Maintainability.** Hard to quantify and usually overlooked dimension of the ETL process quality, which consequently increases the later development cost and the overall ETL project cost in general. Conceptual and logical design enable better documented ETL processes while many ETL tools as well integrate ETL process documentation to allow better maintainability of the project. Potential metrics for the maintainability dimension are the *size* of an ETL process (number of operations and the number of data sources) and the modularity (e.g., atomicity of its operations).

6.5 Architectural setting

Data flow vs. control flow. As previously explained a proper ETL processes design assumes two levels of design, a data flow and a control flow.

- **Data flow** is in charge of performing operations over data themselves in order to prepare them for loading into a DW (or directly for exploiting them by end users). That is, data extraction (reading from the data sources), various data transformation tasks (data cleaning, integration, format conversions, etc.) and finally loading of the data to previously created target data stores of the DW. Data flows are typically executed as a pipeline of operations (rather than a set of strictly sequential steps).

- **Control flow** on the other side is responsible of orchestrating the execution of one or more data flows. It does not work directly with data, but rather on managing the execution of data processing (i.e., scheduling, starting, checking for possible errors occurred during the execution, etc.). Unlike data flow, in control flow the order of execution is strictly defined by the sequence of activities, meaning that one activity does not start its execution until all its input activities have finished. This is especially important in the case of dependent data processing, where the results of one data flow are needed before starting the execution of another.

Staging area. Typically, as a complement to the ETL tool, we need to devise a staging area where to place temporal files. This facilitates *recoverability*, backup of the processes and *auditing*. The purpose of this area is only to support the ETL processing and can contain from plain files to more complex Relational tables, through XML or JSON.

6.6 ETL operations

As summarized in Fig. 6.3, we can find many different operations in ETL tools. Even if they take many different names depending on the tool, they offer simmilar functionalities. Just to hightlight some in Talend terminology:

tMap allows to derive some new columns from existing ones (e.g., it merges two columns into a single one), field renaming and projecting out some columns.

tUniqRow assumes the input is ordered and removes duplicates.

tSortRow orders the rows according to some criteria.

tAggregateRow assumes the input is sorted and generates groups of rows to be aggregated.

tFilterRow obviously filters out those rows that are not interesting.

tJoin is a very expensive operation, which consequently has different implementations to chose.

tUnite is actually a union eliminating duplicates.

tConvertType is a useful operator that allows datatype conversion.

tFileInputDelimited and tDBInput operations allow to connect to different sources, either CSV files or Relational tables.

tDBOutput operations allow to store the result of the flow in some repository. Available types use to coincide with those in the extraction (from CSV files to Relational tables).

We usually refer to the ETL flow as a pipe, in the sense that rows go through it one after another non-stop. Nevertheless, this is not actually true for all operations. Only some of them (known as *non-blocking*) really allow that behaviour (e.g., Field Value Alteration, Single Value Alteration, Sampling, Dataset Copy, Duplicate Row, Router, Union, Field Addition, Datatype Conversion, Field Renaming, Projection, Pivot, Extraction, Loading), while the others actually need to hold all the rows back (i.e., they *block* the flow) until the get the last one (e.g., Duplicate Removal, Sort, Aggregation, Join, Intersect, Difference, Unpivot).

Multimedia Materials

ETL Definition and Motivation (en)

ETL Definition and Motivation (ca)

ETL Steps (en)

ETL Steps (ca)

ETL Process Design (en) 

ETL Process Design (ca) 

Architectural Setting (en) 

Architectural Setting (ca) 

ETL Operations (en) 

ETL Operations (ca) 

Operation Level	Operation Type	Pentaho Data Integration	Talend Data Integration	SSIS	Oracle Warehouse Builder
Attribute	Attribute Value Alteration	Add constant Formula Number ranges Add sequence Calculator Add a checksum	tMap tConvertType tReplaceList	Character Map Derived Column Copy Column Data Conversion	Constant Operator Expression Operator Data Generator Transformation Mapping Sequence
Dataset	Duplicate Removal	Unique Rows Unique Rows (HashSet)	tUniqRow	Fuzzy Grouping	Deduplicator
	Sort	Sort Rows	tSortRow	Sort	Sorter
	Sampling	Reservoir Sampling Sample Rows	tSampleRow	Percentage Sampling Row Sampling	
	Aggregation	Group by Memory Group by	tAggregateRow tAggregateSortedRow	Aggregate	Aggregator
	Dataset Copy		tReplicate	Multicast	
Entry	Duplicate Row	Clone Row	tRowGenerator		
	Filter	Filter Rows Data Validator	tFilterRow tMap tSchemaComplianceCheck	Conditional Split	Filter
	Join	Merge Join Stream Lookup Database lookup Merge Rows Multiway Merge Join Fuzzy Match	tJoin tFuzzyMatch	Merge Join Fuzzy Lookup	Joiner Key Lookup Operator
	Router	Switch/Case	tMap	Conditional Split	Splitter
	Set Operation - Intersect	Merge Rows (diff)	tMap	Merge Join	Set Operation
	Set Operation - Difference	Merge Rows (diff)	tMap		Set Operation
	Set Operation - Union	Sorted MergeAppend streams	tUnite	Merge Union All	Set Operation
Schema	Attribute Addition	Set field value Set field value to a constant String operations Strings cut Replace in string Formula Split Fields Concat Fields Add value fields changing sequence Sample rows	tMap tExtractRegexFields tAddCRCRow	Derived Column Character Map Row Count Audit Transformation	Constant Operator Expression Operator Data Generator Mapping Input/Output parameter
	Datatype Conversion	Select Values	tConvertType	Data Conversion	Anydata Cast Operator
	Attribute Renaming	Select Values	tMap	Derived Column	
	Projection	Select Values	tFilterColumns		
Relation	Pivoting	Row Denormalizer	tDenormalize tDenormalizeSortedRow	Pivot	Unpivot
	Unpivoting	Row Normalizer Split field to rows	tNormalize tSplitRow	Unpivot	Pivot
Value	Single Value Alteration	If field value is null Null if Modified Java Script Value SQL Execute	tMap tReplace	Derived Column	Constant Operator Expression Operator Match-Merge Operator Mapping Input/Output parameter
Source Operation	Extraction	CSV file input Microsoft Excel Input Table input Text file input XML Input	tFileInputDelimited tDBInput tFileInputExcel	ADO .NET / DataReader Source Excel Source Flat File Source OLE DB Source XML Source	Table Operator Flat File Operator Dimension Operator Cube Operator
Target Operation	Loading	Text file output Microsoft Excel Output Table output Text file output XML Output	tFileOutpu tDelimited tDBOutput tFileOutputExcel	Dimension Processing Excel Destination Flat File Destination OLE DB Destination SQL Server Destination	Table Operator Flat File Operator Dimension Operator Cube Operator

Figure 6.3: ETL operations [The17]

Chapter 7

Data Quality

Following from the general definition of quality, the data is considered of high quality if it is *fit for its intended use* [BS16], meaning that the level of quality to be considered as acceptable, depends on the real needs of end user, and the purpose of the underlying data. Containing errors in the patient medical records, like happened in 2010 in UK,[1] is simply unacceptable, since people's life depends on their use. In the case of Machine Learning, [BPR+19] empirically proves that the performance of the resulting model is bounded by the quality of the underlying training data.

Data quality is sometimes wrongly reduced just to *accuracy* (e.g., name misspelling, wrong birth dates). However, other dimensions such as *completeness*, *consistency*, or *timeliness* are often necessary in order to fully characterize the quality of information.

7.1 Sources of problems in data

Different data quality problems may be introduced throughout the entire data pipeline.

- **Data ingestion.** While data being initially retrieved, accuracy issues can be introduced due to faulty *data conversions* or simply by making typos in the *manual data entry*. Similarly, *data timeliness* can also be affected if the data are being fetched periodically in the form of *batch feeds*, while being consumed through *real-time interfaces* implies expecting high *freshness* of the data. If data are being fetched from *consolidating various systems*, *consistency* errors may occur due to the heterogeneity of these source systems.

- **Data processing.** New errors can also be introduced while data are being processed for their final use. For instance, *process automation* may lead to overlooking some special cases and thus applying a default processing that result in inaccurate outputs (e.g., a missing case in a conditional statement in the code can lead to a default option). Paradoxically, new accuracy issues and other data imperfections can also be introduced during *data cleansing* actions (e.g., replacing missing values with default constants or precalculated aggregates like means or mediums). Finally, we can also affect the *completeness* and *consistency* of our data due to incautious *data purging* actions, by mistakenly deleting some of data values.

- **Inaction.** Lastly, data quality can also be largely affected by not taking necessary actions to prevent their occurrence. For instance, our data can become *inconsistent* if the *changes* that occur at one place are not properly propagated to the rest of the data that relates to it. Also, various *inaccuracies* can occur if the new *system upgrades* are not properly controlled, especially those that lack backward compatibility (e.g., data types or operations being changed or removed from DBMS). Generally, the same data that previously were considered of high quality can become flawed if the *new use of data* is considered (see the definition of quality above as data fitting their intended use).

[1] https://www.thetimes.co.uk/article/new-patient-medical-records-database-contains-life-threatening-errors-2f5b28320m7

7.2 Data Conflicts

Data conflicts refer to the deviations between data capturing the same real-world entity [Do]. Cleaning of data with conflicts is required in order to avoid misleading and faulty data analysis.

7.2.1 Classification of Data Conflicts

Data conflict can be classified from two orthogonal perspectives: (a) based on the level at which they occur (i.e., *schema* vs. *instances*), and (b) if they occur within a *single source* or among *multiple sources* (see Figure 7.1).

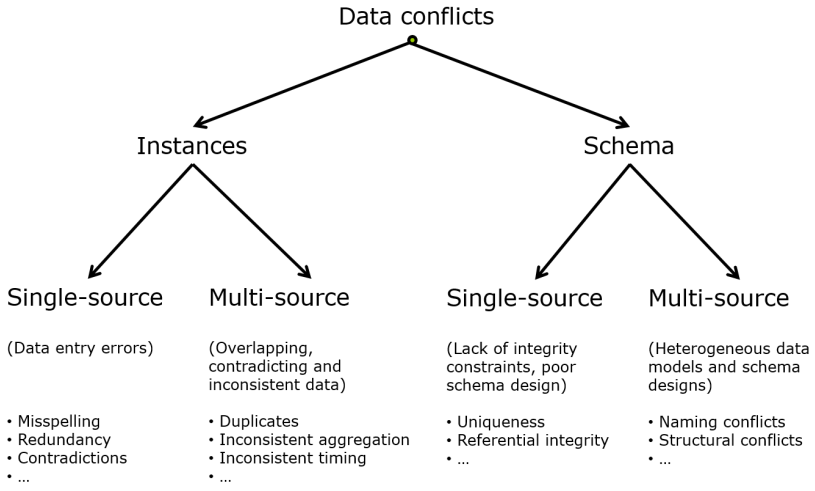


Figure 7.1: Data conflicts classification

Schema vs. instances. *Schema-level conflicts* are caused by the errors in the design of the data schemata (e.g., duplicates caused by the lack of unique constraint, inconsistency due to mistake in the referential integrity, structural schema conflicts among various data sources). *Instance-level conflicts* refer to errors in the actual data contents, which are typically not visible nor preventable at the schema design time (e.g., misspelling, redundancy or data duplicates and inconsistencies among various data sources). Both schema- and instance-level conflicts can be further differentiated based on the scope at which they occur: within an individual *attribute*, between attributes of a single *record*, between different records of certain *record type*, and between different records of different record type. **Single- vs. Multi-source.** The number of *single-source data* conflicts obviously largely (but not only) depends on the degree to which it is governed by schema constraints that control permissible data values. Therefore, the data sources that do not have native mechanisms or are more flexible to enforce a specific schema (e.g., CSV files, JSON documents) are more prone to errors and inconsistencies. All the errors potentially present in a single source are aggravated when *multiple sources* need to be integrated. Data may already come with errors from single sources, but they may additionally be represented differently, may overlap or even contradictory among various data sources. Problems that are typically caused by the fact that these source systems are independently developed.

7.2.2 Dealing with data conflicts

Conflicts in data can be either *prevented* from occurring or *corrected* once they occur. Obviously, preventing data conflicts is preferable due to the typically high costs of data cleaning. This requires proper schema design with well-defined data constraints or strict enforcement of constraints (e.g., by limiting user errors in the graphical user interface). However, in some cases, and almost always for the case of multiple sources, a posteriori data cleaning actions are necessary to guarantee a required level of data

quality for a particular analysis. Such actions may involve very complex set of data transformation and are typically part of extract-transform-load (ETL) processes (see Chapter 6).

7.3 Data quality dimensions and measures

As already discussed before, data quality is a multidimensional or multifacet problem. The framework proposed in [BS16] classifies data quality in clusters of dimensions, where dimensions are grouped based on their similarity, and the representative (first) dimension gives a name to the cluster.

1. **Completeness**, pertinence, and relevance refer to the capability of representing all and only the relevant aspects of the reality of interest.
2. **Accuracy**, correctness, validity, and precision focus on the adherence to a given reality of interest.
3. **Redundancy**, minimality, compactness, and conciseness refer to the capability of representing the aspects of the reality of interest with the minimal use of informative resources.
4. **Readability**, comprehensibility, clarity, and simplicity refer to ease of understanding and fruition of information by users.
5. **Accessibility** and availability are related to the ability of the user to access information from his or her culture, physical status/functions, and technologies available.
6. **Consistency**, cohesion, and coherence refer to the capability of the information to comply without contradictions to all properties of the reality of interest, as specified in terms of integrity constraints, data edits, business rules, and other formalisms.
7. **Usefulness**, related to the advantage the user gains from the use of information.
8. **Trust**, including believability, reliability, and reputation, catching how much information derives from an authoritative source. The trust cluster encompasses also issues related to security.

In what follows, we present in more details the four most relevant data quality dimensions, i.e., *completeness*, *accuracy*, *timeliness* (a special case of temporal accuracy), and *consistency*, and for each dimension we present as well a quality measure formula used to estimate the achievement of such data quality dimension within the dataset [Sat18]. An study of the impact of some of these measures in the performance of ML-models can be seen in [BFI+22].

7.3.1 Completeness

Data completeness is most typically related to the issue of missing or incomplete data, and as such it is considered one of the most important data quality problems.

Most often interpretation of the data completeness is the absence of *null* values, or more exactly, the ratio of non-null values and the total number of values. Such interpretation can be easily represented as a measure. First, at the level of a single attribute (A_i) we have:

$$Q_{Cm}(A_i) = \frac{|R(NotNull(A_i))|}{|R|} \tag{7.1}$$

Following from there, the completeness of the entire Relation (R) can be measured by counting the number of tuples containing no null values. That is:

$$Q_{Cm}(R) = \frac{|R(\wedge_{A_i \in R} NotNull(A_i))|}{|R|} \tag{7.2}$$

However, the problem of measuring the completeness in this way is that *null* can have different semantics in our dataset. It could really mean a missing value or simply a not-applicable case (e.g., a customer without a special delivery address).

Moreover, the absence of values for different attributes may have higher or lower importance. For instance, while we typically always need an identifier attribute (e.g., customer ID), having the city of birth missing may be optional. To address such a case, we can assign corresponding weights to the calculation of the completeness of each attribute in the Relation.

Note these measures compute the completeness of the data that are inside the database, i.e., following the Closed World Assumption (CWA). However, the original definition expresses the completeness as the degree to which a given dataset describes the corresponding set of real-world objects. In this case, measuring the completeness can be more difficult, as it may require additional metadata or cross checking with real world (e.g., possibly by sampling). Lastly, another aspect of completeness refers to the fact if a real-world property is represented as an attribute in a dataset or not. However, assessing this aspect as well requires (often manual) inspection with the real world.

7.3.2 Accuracy

The quality of data can as well be affected by measurement errors, observation biases, or improper representations. This aspect is covered by the accuracy dimension, which can be defined "as the extent to which data are correct, reliable, and certified free of errors". The meaning of accuracy is application-dependent, and it can be expressed as the distance from the actual real-world value or the degree of detail of an attribute value. For instance, a product sales volume variable in our data warehouse can have a value 10.000$, while the value in reality (e.g., in accounting system) is 10.500$. While in general this can be interpreted as inaccurate, in another use case, where the user is only interested in binary sales categories (low: $\leq$ 20.000$, high: $>$ 20.000$), the value in the data warehousing system can be considered as accurate enough.

To assess the accuracy in a general case, we need to calculate the distance of the value stored in a our database from the actual real value.

$$e_A = |V_a - V_{RealWorld}| \tag{7.3}$$

In the case of numerical attribute such distance is simply an arithmetic difference, while in the case of other data types it may require more complex way of calculation. For example, *Hamming distance* calculates the number of positions at which two strings of the same length differ, or its more complex version, *Levenshtein (or Edit) distance*, that calculates the minimal number of character-level operations (insertions, deletion or substitutions) required to change one string into another one.

Regardless of the way e_A is calculated, we further assess the accuracy of an attribute as follows.

$$Q_A(A_i) = \frac{|R(e_{A_i} \leq \varepsilon)|}{|R|} \tag{7.4}$$

Notice that the threshold ε is application specific and it determines the level of tolerance for the accuracy of a certain attribute.

As before, we can further apply this to the whole Relation as follows:

$$Q_A(R) = \frac{|R(\wedge_{A_i \in R} e_{A_i} \leq \varepsilon)|}{|R|} \tag{7.5}$$

However, in some cases, we may need to adopt more complex means for assessing the accuracy. For instance, for textual attributes we may need to consider *semantic distance* of their values (e.g., both Munich and München represent the same city although they are syntactically "far from each other"). Resolving such a problem, would require the existence of a dictionary or ontology mapping.

7.3.3 Timeliness

Another problem that data may suffer is that they are outdated, meaning that they are captured/extracted from the original source potentially before the new changes in the source may occurred. This dimension is also referred to as *freshness*. The level of timeliness as well depends on the specific application that the data is used for. For example, the last year's accounting data may be considered as "old", but if the users are actually auditing company's last year's operations, these data is just what they need.

On the other hand, the real "age" of the data is also determined by the *frequency* of updates that may occur over the data since the last capturing/extraction. For example, the system that uses UN's population and migration indicators may have data that are months old, but they may be considered as "fresh" having that the UN publishes these indicators annually.

We thus need to consider both value's *age* (i.e., calculated from the time when the datum is committed in the database, a.k.a. Transaction Time[2], as $age(v) = now - transactionTime$) and its *frequency of updates* per time unit ($f_u(v)$) in order to assess its timeliness.

$$Q_T(v) = \frac{1}{1 + f_u(v) \cdot age(v)} \tag{7.6}$$

We can further extend this to an attribute (A_i), as:

$$Q_T(A_i) = Avg_{v \in R[A_i]} Q_T(v) \tag{7.7}$$

, and to the entire Relation (R), as:

$$Q_T(R) = Avg_{A_i \in R} Q_T(A_i) \tag{7.8}$$

The consequence of this is that data with higher update frequency "ages" faster while those that are never updated are always considered as "fresh".

7.3.4 Consistency

Consistency refers to the degree to which data satisfies defined semantic rules. These rules can be integrity constraints defined in the DBMS (i.e., *entity* - "a primary key of a customer cannot be null", *domain* - "customer age column must be of type integer", *referential* - "for each order a customer record must exists"), or more complex business rules describing the relationship among attributes (e.g., "a patient must be female in order to have attribute pregnant set to true"), typically defined in terms of *check constraints* or *data edits*[3]. Recent research approaches also propose automatic derivation of such rules from the training data by applying rule induction.

Having that B is a set of such rules for Relation R, we can calculate the ratio of tuples that satisfy all the rules as:

$$Q_{Cn}(R, B) = \frac{|R(\wedge_{rule \in B} rule(A_1, .., A_n))|}{|R|} \tag{7.9}$$

7.3.5 Trade-offs between data quality dimensions

Data quality dimensions often cannot be considered in an independent manner, given that there is a correlation between some of them [BS16]. Therefore, favoring one data quality dimension may have negative consequence for the other ones. We distinguish two important cases.

1. *Timeliness* $\Longleftrightarrow$ (*Accuracy*, *Completeness*, and *Consistency*). Having accurate (or complete or consistent) data may require either checks or data cleaning activities that take time and thus can affect negatively the timeliness dimension. Conversely, timely data may suffer from accuracy (or completeness or consistency) issues as their freshness was favored. Such trade-offs must always be made depending on the application. For instance, in some Web applications, timeliness is often preferred to accuracy, completeness and consistency, meaning that it is more important that some information arrives faster to the end user, although it initially may have some errors or not all the fields be completed.

2. *Completeness* $\Longleftrightarrow$ (*Accuracy* and *Consistency*). In this case, the choice to be made is if it is "better" (i.e., more appropriate for a given domain), to have less but accurate and consistent data, or to have more data but possibly with errors and inconsistencies. For instance, for social statistical

[2] https://en.wikipedia.org/wiki/Transaction_time
[3] https://en.wikipedia.org/wiki/Data_editing

analysis it is often required to have a significant and representative input data, thus we would favor completeness over accuracy and consistency in such case. Conversely, when publishing scientific experiment results, it is more important to guarantee their accuracy and consistency than their completeness.

7.4 Data quality rules

As discussed before, data quality conflicts can be either prevented a priori or corrected a posteriori. Relational DBMSs offer various mechanisms to prevent such conflicts by defining data quality rules in terms of *integrity constraints* or *dependencies*. However, such rules need to satisfy a set of *logic properties* in order to guarantee their minimality and that they can accommodate the data (i.e., that there is an instance of a dataset satisfying all the rules).

7.4.1 Integrity constraints and dependencies

Integrity constraints can be further classified as follows:

- **Intra-Attribute.** Constraints that affect a single database column.
  - *Domain.* The set of values permitted for a given column. This is typically defined by the column data type, but can as well be more complex (e.g., to detect column's outliers).
  - *Not null.* To guarantee the existence of the value for such a column, and thus improve data *completeness* dimension.
- **Intra-Tuple.** Constraints that affect single tuples (i.e., among different columns of a single tuple).
  - *Checks* are the most common way of defining the intra-tuple constraints in a Relational database. For instance, "`basicSalary > 2000`" in the *department* table or "`originAirport <> destinationAirport`" in the *flight* table.
- **Intra-Relation.** Constraints that have as a scope the entire Relation (i.e., entire database table).
  - *Primary key* or *Unique* are the most common intra-Relation constraints defined over a database table that guarantee uniqueness (i.e., no repetition of values) of a column or a set of columns in a table.
  - *Temporal* intra-Relation constraints are typically defined by means of database *triggers* and assume the existence of temporal attributes. They may be used to express several rules, such as (being X and Y potentially different for each object): (1) *Currency:* each object must have at least one value for the temporal attribute in the last X days; (2) *Retention:* no object can have a value before data X; (3) *Granularities:* the time distance between two consecutive values of the temporal attribute must be between X and Y; (4) *Continuity:* if the temporal attribute has begin and end timestamps, these intervals cannot overlap and must be adjacent. More complex rules can also be defined. For example, we could check the distance of each point to the linear regression of the temporal attribute.
- **Inter-Relation.** Constraints that define the relationship between the columns of two different database tables.
  - *Foreign key* is the most typical way used to define the referential integrity constraint among two tables. That is, the values of a foreign key column(s) of one table must match some value in the primary key column(s) in another table.
  - *Assertions* can be used to define more complex inter-Relation constraints, not limited to foreign-primary key relationships. Such rules are defined in terms of database triggers that are fired (with the corresponding action execution) when a rule is violated.

Using these integrity constraints we can maintain the following dependencies among the attributes of our database.

- **Multivalued dependency** (whose special case is *functional dependency*) defines that a certain set of attributes X unequivocally determines a value (for functional dependency) or a set of values (in a general case) of another set of attributes Y, of the same Relation. This intuitively means that two tuples sharing the same values for attributes in X must have the same values for the attributes in Y (i.e., $t1.X = t2.X \Rightarrow t1.Y = t2.Y$).
- **Inclusion dependency** is maintained by defining the referential integrity constraint (*foreign key*) between the attributes of two database tables. Intuitively, if Y is a set of attributes that defines a primary key of table S and X is a set of attributes that defines a foreign key in table R that references $S.Y$, it is gurauranteed that a set of tuples over attributes $R.X$ is subsumed by a set of tuples over attributes $S.Y$ (i.e., $R.X \subseteq S.Y$).

7.4.2 Logic properties of data quality rules

However, if not defined with care, the constraints that express the desired data quality rules may yield certain imperfections, subotimalities or even contradictions. For instance, two checks that are defined over the same salary attribute may be contradictory (e.g., $CHECK\ (salary < 1000\ AND\ salary > 2000)$) and hence generate empty empty results (i.e., prevent generating any tuple). Similarly, a rule may also be redundant to another previously defined rule (e.g., $CHECK(salary > 2000)$ and $CHECK(salary > 1000)$), meaning that the latter rule will be impossible to violate, hence its checking introduces an unnecessary overhead.

To guarantee that a set of data quality rules defined over a database is minimal and allows tuple generation, we can rely on the following set of *logic properties of rules*. Notice that these logic properties are defined over different database elements (schema, tables, views, constraints, tuples, queries), depending on their scope and semantics.

- **Schema-satisfiability**: A *schema* is satisfiable if there is at least one consistent DB state containing tuples (i.e. each and every constraint is fulfilled).
- **Liveliness**: A *table/view* is lively if there is at least one consistent DB state, so that the table/view contains tuples.
- **Redundancy**: An *integrity constraint* is redundant if the consistency of the DB does not depend on it (none of the tuples it tries to avoid can never exist).
- **State-reachability**: A given *set of tuples* is reachable if there is at least one consistent DB state containing those tuples (and maybe others).
- **Query containment (subsumption)**: A *query* Q1 is contained in another Q2, if the set of tuples of Q1 is always contained in the set of tuples of Q2 in every consistent DB state.

7.4.3 Fine-tuning data quality rules

As we discussed earlier, data quality rules, may yield certain imperfections when it comes to data quality assessment. Indeed, we can find both situations where data quality rules fall short, these being *false positives* and *false negatives* [May07]. The former refers to the well known cases when a rule detects as erroneous tuples that in reality are correct, while in the latter case, the rule fall short in detecting an erroneous tuple. These rule imperfections are very important for accurately assessing the quality of data and thus require further fixing and fine-tuning in order to enhance the rules as much as possible. A process of rule fine-tuning proposed in [May07] is depicted in Figure 7.2.

The process starts by identifying rule imperfections, followed by the analysis of such findings and searching for the patterns in the data. In the third step, the rules are enhanced to eliminate as many imperfections as possible. These three steps are executed iteratively until we are satisfied with the enhancement or we simply run out of the resources. Such process largely relies on manually verifying

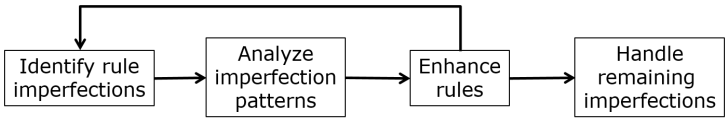


Figure 7.2: Steps in rule fine-tuning process [May07]

samples of real-world data and the comparison with the errors identified by applying rules. In the final step, we handle the remaining imperfections, i.e., those that could not be improved but we are at least aware of and are accounted for in the data quality assessment.

7.5 Data quality improvement

As we said, data quality rules can be used to prevent data conflicts from occurring in our datasets. However, and especially in the case when data are coming from external and heterogeneous data sources, it is impossible to prevent errors in data a priori. In such cases, when data conflicts are detected, we need to proceed (a posteriori) with their corrections in order to improve the data quality in the resulting dataset. Such process of restoring correct values is commonly known as *imputation*. Imputations should be done following the two main goals [FH76]: (1) *minimum change principle*, stating that the corrections on data must be done by changing the fewest values possible, and (2) maintain the marginal and joint frequency distribution of values in the different attributes. However, these two goals may be in conflict. For example, a tuple <6, married, retired> of table <age, maritalStatus, typeOfWork> is detected as erroneous given the previously defined data quality rule ($maritalStatus = married \Rightarrow age \geq 15$). To correct this and similar tuples with minimum change, we can impute a value 15 to any value violating the given rule. However, by doing so, we alter the *marginal distribution* of values of age. Even if we perform imputation respecting the distribution of values of age, we may affect the *joint distribution* with attributes maritalStatus and typeOfWork. Following from this, we can see that data quality improvement requires more complex and wider set of changes.

Generally, the methodologies proposed for improving the quality of data adopt two general types of strategies, i.e., *data-driven* and *process-driven* [BS16]. *Data-driven* strategies improve the quality of data by directly modifying the value of data. For example, obsolete data values are updated by refreshing a database with data from a more current database. *Process-driven* strategies improve quality by redesigning the processes that create or modify data. As an example, a process can be redesigned by including an activity that controls the format of data before storage.

Both data-driven and process-driven strategies apply a variety of techniques: algorithms, heuristics, and knowledge-based activities, whose goal is to improve data quality. In what follows, we present a (non complete) list of data-driven data quality improvement techniques:

1. **Acquisition of new data**, which improves data by acquiring higher-quality data to replace the values that raise quality problems;

2. **Standardization** (or **normalization**), which replaces or complements nonstandard data values with corresponding values that comply with the standard. For example, nicknames are replaced with corresponding names, (e.g., Bob with Robert), and abbreviations are replaced with corresponding full names, (e.g., Channel Str. with Channel Street);

3. **Entity resolution** (or **record linkage**), which identifies if the data records in two (or multiple) tables refer to the same real-world object;

4. **Data and schema integration**, which define a unified view of the data provided by heterogeneous data sources. Integration has the main purpose of allowing a user to access the data stored by heterogeneous data sources through a unified view of these data. Data integration deals with quality issues mainly with respect to two specific activities:

- *Quality-driven query processing* is the task of providing query results on the basis of a quality characterization of data at sources.
- *Instance-level conflict resolution* is the task of identifying and solving conflicts of values referring to the same real-world object.

5. **Source trustworthiness**, which selects data sources on the basis of the quality of their data;
6. **Error localization** and **correction**, which identify and eliminate data quality errors by detecting the records that do not satisfy a given set of quality rules. These techniques are mainly studied in the statistical domain. Compared to elementary data, aggregate statistical data, such as average, sum, max, and so forth are less sensitive to possibly erroneous probabilistic localization and correction of values. Techniques for error localization and correction have been proposed for inconsistencies, incomplete data, and outliers.
7. **Cost optimization**, defines quality improvement actions along a set of dimensions by minimizing costs.

7.6 Object identification

Object identification is the most important and the most extensively investigated information quality activity [BS16], and it refers to the process of identifying whether the data in the same or in different data sources represent the same object in the real world.

Table 7.1: Example of three national registries (agencies) that represent the same business [BS16]

Agency	Identifier	Name	Type of activity	Address	City
Agency 1	CNCBTB765SDV	Meat production of John Ngombo	Retail of bovine and ovine meats	35 Niagara Street	New York
Agency 2	0111232223	John Ngombo canned meat production	Grocer's shop, beverages	9 Rome Street	Albany
Agency 3	CND8TB76SSDV	Meat production in New York state of John Ngombo	Butcher	4, Garibaldi Square	Long Island

Example in Table 7.1 shows that the same object in the real world (i.e., the same business) can be represented in different manners in three national registries (i.e., *Agency 1*, *Agency 2*, and *Agency 3*). Some differences like different identifiers may be simply due to the different information systems from where these tuples are coming from. Other attributes like name, type of activity, address, and city also differ (although with some similarities), and this can be due to several reasons, like typos, deliberately false declarations, or data updated at different times.

The high-level overview of the *object identification process* is depicted in Figure 7.3. Assuming for simplicity two input data sources (A and B), the process includes the following activities:

1. **Pre-processing** has as a goal to normalize/standardize the data and correct evident errors (e.g., conversion of upper/lower cases) and reconcile different schemata.
2. **Search space reduction.** Performing entity resolution on the entire search space (i.e., Cartesian product of tuples in inputs) would result in complexity $O(n^2)$, n being the cardinality of input Relations. To make this process tractable we first need to reduce the given search space. This is typically done by means of three different methods: (a) *blocking*, which implies partitioning a file into mutually exclusive blocks and limiting comparisons to records within the same block, (b) *Sorted neighborhood* consists of sorting a file and then moving a window of a fixed size on the file, comparing only records within the window, and (c) *Pruning* (or *filtering*) implied removing from the search space all tuples that cannot match each other, without actually comparing them.

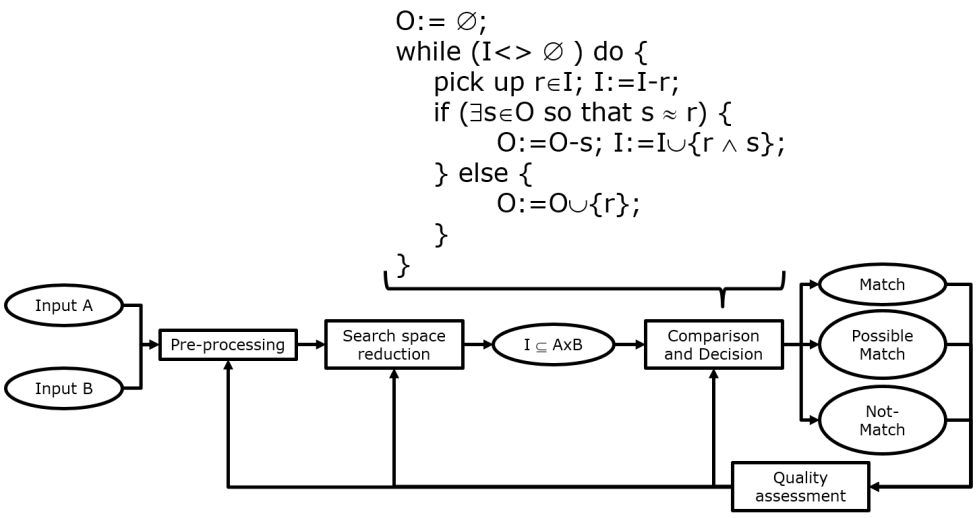


Figure 7.3: Relevant steps of object identification techniques [BS16]

3. **Comparison and decision.** This step includes an entity resolution (or record linkage) technique that first selects a *comparison function* used to compare if two tuples represent the same object in the real world, by calculating the "distance" between them (see Section 8.3.4.1 for more details), and then *decides* if the compared tuples *match*, *do not match*, or *potentially match* based on the analysis of the results of the comparison function. An example of the entity resolution algorithm (R-Swoosh [GUW09]) is presented in Figure 7.3, while more details are provided in Section 8.3.4.3. It may also happen that no decision can be made automatically and a domain expert has to be involved to make the decision.

4. **Quality assessment** is finally performed based on the result of the previous comparison and data quality indicators are measured to assess if the results are satisfactory. Minimizing *possible matches* is a typical goal to avoid as much as possible the involvement of the domain expert. In addition, minimizing *false positives* and *false negatives* are as well common goals in the quality assessment step.

Multimedia Materials

Motivation and Definition (en)

Motivation and Definition (ca)

Data Quality Measures (en)

Data Quality Measures (ca)

Data Quality Rules (en)

Data Quality Rules (ca)

Data Quality Improvement (en)

Data Quality Improvement (ca)

Chapter 8

Schema and data integration

With contributions of Fernando Mendes Stefanini and Oscar Romero

The integration problem appears when a user needs to be able to pose one query, and get one single answer, so that in the preparation of the answer data coming from several DBs is processed. It is important to notice that the users of the sources coexist (and should be affected as little as possible) with this new global user. In order to solve this, we can take three different approaches:

a) Manually query the different databases separately (a.k.a. *Superman* approach), which is not realistic, since the user needs to know the available databases, their data models, their query languages, the way to decompose the queries and how to merge back the different results.

b) Create a new database (a.k.a. DW or ERP) containing all necessary data.

c) Build a software layer on top of the data sources that automatically splits the queries and integrates the answers (a.k.a. *federation* or *mediation* approach).

8.1 Distributed Databases

In the end, independently of the approach we take, we are talking about a distributed database but with a variable degree of autonomy in the components. This autonomy impacts the design (components are designed independently), the execution (components can execute in different ways and offering different guarantees), and the association (components can easily and freely leave the system).

8.2 Semantic heterogeneities

All this autonomy generates heterogeneities in the end. In what follows, we first discuss the semantic heterogeneities that may appear among the autonomously built data sources.

8.2.1 Definitions

Some terms like *classes*, *entities*, *attributes* can have different meanings depending on the context they are applied. In order to avoid ambiguities, we first define how some common terms are going to be employed. With this, we hope to aid explanations in further sections.

8.2.1.1 Concepts

In the context of this chapter, the term *class* and *entity* is used interchangeably to define a concept of the domain, analogously to a Relational table in Relational databases or a class in object oriented designs, such class is composed by attributes and might have relationship with other classes. Thus, we define class C as a concept containing a set of attributes. Any concept can be represented by a class in a

UML Class diagram. For example, the class `Playlist` containing attributes `name, lastModifiedData, Description, #Followers`; and the class `Track` containing attributes `trackName, artistName, note`; can be represented as:

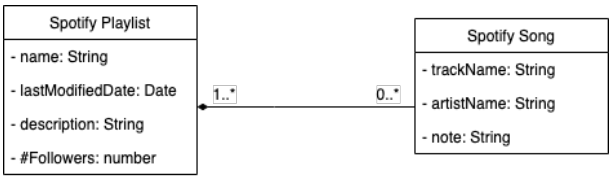


Figure 8.1: Sample of Spotify classes.

This representation is conceptual and agnostic to the way the data are physically stored. To illustrate that, we display an snippet of the file `Playlists.json` from Spotify, which contains the data from which the diagram in Figure 8.1 was generated. The full conceptual model for a given data source is then called the *schema*.

Listing 8.1: Snippet of File `Playlist.json`

```
{
"playlists": [
{
"name": "Bjork Pitchfork",
"lastModifiedDate": "2018-10-16",
"tracks": [
{
"trackName": "Cosmogony",
"artistName": "Bjork",
"note": "Usually the first song of the concert"
},
{
"trackName": "Hunter",
"artistName": "Bjork",
"note": null
},
...],
"description": null,
"numberOfFollowers": 0
},
...]
}
```

Lastly, an *instance* will be an instantiation of a concept of the schema; i.e.: the playlist "Bjork Pitchfork" is an instance of the class playlist.

8.2.1.2 Equivalence and Hierarchy

Furthermore, we elaborate in the notation used to express relationship between classes, and attributes, that will be further used in the rest of the work. Given a set of classes $\{C_1, C_2, ..., C_n\}$ we say that:

1. $C_1 \equiv C_2$ iff C_1 represents the same concept as C_2 (equivalent).
2. $C_1 \sqsubset C_2$ iff C_2 represents a broader concept that subsumes C_1 (C_1 subclass of C_2 / C_2 superclass of C_1).

8.2.2 Classification of semantic heterogeneities

In order to integrate the data from various data sources, we face the problem of dealing with the heterogeneity between the two schemas, their concepts and their instances. Thus, we formalize the possible heterogeneities between data sources by splitting them in two main groups, *Intra-Class heterogeneity* and *Inter-Class heterogeneity*; some naming and definitions were inspired on the work of [GSSC95] . We use

C_B to represent the set of classes $\{C_{B1}, C_{B2}, ..., C_{Bn}\}$ from one domain schema (represented in blue), and C_G to represent the set of classes $\{C_{G1}, C_{G2}, ..., C_{Gm}\}$ of another domain schema (represented in green).

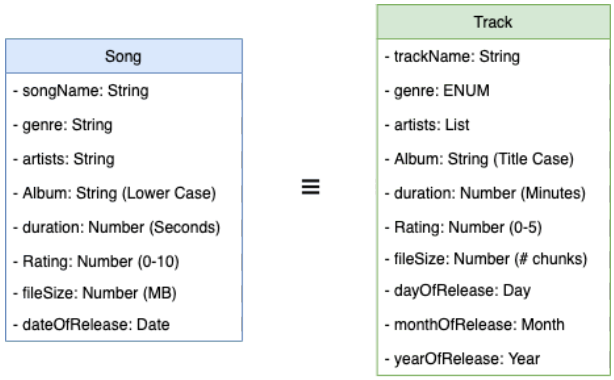


Figure 8.2: $C_{B1} \equiv C_{G1}$ where C_{B1} = Song and C_{G1} = Track

8.2.2.1 Intra-Class Heterogeneity

We classify as intra-class heterogeneity the cases where there is a class in one schema that is equivalent to a class in the other, i.e.: $C_{B1} \equiv C_{G1}$. This means that both classes represent the same concept, but because of the characterization of the concepts in each schema, some discrepancies can be found both within the classes and within their attributes. We take for example the case in Figure 8.2, where song from the C_B and track from the C_G are equivalent concepts. We consider the intra-class heterogeneities in Table 8.1.

Class	Naming		When the classes have different names (e.g., Song in C_B and `Track` in C_G).
Attribute	Naming		When the attributes have different names (e.g., `songName` and `trackName`).
	Type		When two equivalent attributes have different type representation between domains (e.g., `genre` in C_B and in C_G).
	Single/Multi-valuation		When the attribute is single-valued in one domain but multi-valued in another (e.g., `artists` in C_B and C_G).
	Representation	Format	When the attributes have the same type, but are represented in different formats (e.g., `Album` in C_B and C_G).
		Measure	When two attributes are numerical, but the unit to represent them differs (e.g., `duration` in C_B and C_G).
		Scale	When two attributes are numerical, but the scale to represent them differs (e.g., `rating` in C_B and C_G).
		Dimension	When two attributes are numerical, but the dimension to represent them differs (e.g., `fileSize` in C_B and C_G).
	Composition		When one attribute is the result of the composition of many attributes (e.g., `dateOfRelease` in C_B and `dayOfRelease`, `monthOfRelease` and `yearOfRelease` in C_G).

Table 8.1: Intra-Class heterogeneities

8.2.2.2 Inter-class Heterogeneity

For the cases when we can not do a direct equivalence between two classes, we might need to express it as a composition of one or many classes, or super/sub classes. Thus, we give an example in Figure 8.3 of the classes belonging to C_B and C_G, with the following arrangements between two schemata:

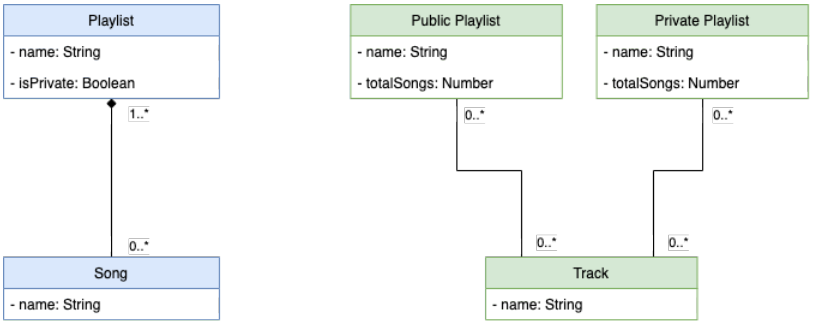


Figure 8.3: C_B and C_G classes.

- Public Playlist ⊏ Playlist
- Private Playlist ⊏ Playlist
- Song ≡ Track

We identify the inter-class heterogeneities that we consider between the two sets of classes in Table 8.2.

Super/Subclass of		When a Class in a schema is a generalization or specification of a class in another (e.g., `Playlist` in C_B and `Public/Private Playlist` in C_G).
Aggregation	Composition	When a class is composed by an aggregation of another class in one domain, but is an independent class in another (e.g., Song is aggregated inside `Playlist` in C_B while `Track` has simple associations with `Public/Private Playlist` in C_G).
	Derivation	When the characterization of a class is derived from the aggregation of another (e.g., `Public/Private Playlist` in C_G have `totalSongs`, an attribute defined by the aggregation of `Track` in C_G. While in C_B such dependency is absent).

Table 8.2: Inter-Class heterogeneities

8.3 Overcoming heterogeneity

Next, we will see how some of these heterogeneities can be resolved by schema and data integration solutions.

8.3.1 Wrappers and mediators

One solution (that belongs to the class of federation/mediation approaches) for overcoming the heterogeneity and resolving data integration problem is a *mediator-based system* [GUW09]. Differently from the DW systems, *mediators* do not store any data, but rather offer an interface for posing a single query, decomposing such query so that several (typically more than 2) data sources are queried and then integrates the results of individual queries in order to return a single unified answer to the user (see Figure 8.4).

Mediators can be seen as virtual views but instead of accessing various database tables, they access various (typically independent) data sources. To access data of each data source, mediators work in combination with another set of components - *wrappers*. A wrapper represents a piece of software that understands the underlying language and format of a data source system, and it can query it in order to retrieve the data back to the mediator which combines the results and returns the answer to the user.

For instance, consider that one data source is an RDBMS and that a wrapper has access to table `vehicles`:

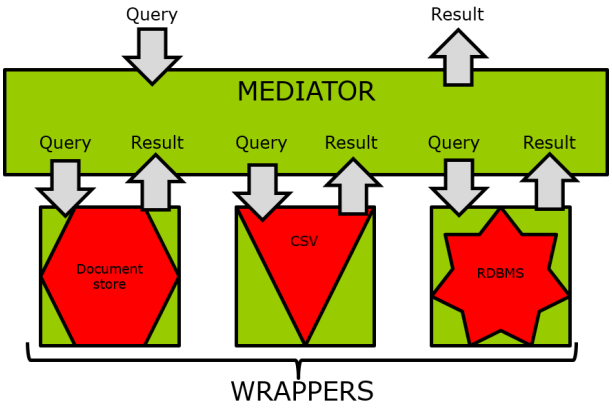


Figure 8.4: Mediator-based system

```
vehicle (serial_n,make,model,auto_transmission,air_conditioning,color)
```

We can implement such wrapper (using the corresponding JDBC driver) in the form of an SQL query template that includes parameters representing constants. For example, a wrapper template to access the vehicles of certain color would be:

```
SELECT serial_n,model,make,auto_transmission, color
FROM vehicle
WHERE color = '$c';
```

In addition, consider that there exists another data source based on CSV files, and with schemata:

```
car(serial_n, make, model, auto_transmission, color)
optional_equipment(serial_num, option, price)
```

In this case, to answer the same previous query, our second wrapper needs to access both CSV files (e.g., using Java file API) and retrieve data from the car file to be programmatically joined (no SQL query is actually generated in this case) with those in optional_equipment, if needed.

Given those two wrappers, a mediator (e.g., implemented in Java), after receiving a global query (over the mediated_car table) needs to forward the query to the corresponding wrapper templates. For example, the mediator would forward a query to the first wrapper like this:

```
SELECT *                          SELECT serial_n,make,model,auto_transmission,color
FROM mediated_car          =>     FROM vehicle
WHERE color = '$c';               WHERE color = '$c';
```

Finally, the mediator combines the results of the two queries and returns the final result to the user.

8.3.2 Major steps to overcome semantic heterogeneity

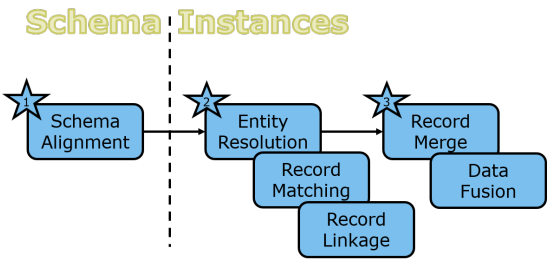


Figure 8.5: Major steps to overcome semantic heterogeneities

For enabling the mediator-based system to automatically handle the heterogeneity among various data sources, there are three major steps to be considered (see Figure 8.5):

1. *Schema alignment* works at the schema level, and includes establishing the correspondences (a.k.a. mappings) among the concepts (i.e., attributes or variables) of different data sources.

2. *Entity resolution*, also know as *record matching* or *record linkage*, is the process focused on finding the instances (i.e., records, tuples or values) that represent the same entities in reality.

3. *Record merge*, also known as *data fusion*, is the final step of data integration, in which, after the equivalent records are found, they are merged, either by combining them or keeping the more "accurate" or more "trustworthy" one.

Schema alignment corresponds to what is know as *schema integration*, while the latter two (*entity resolution* and *record merge*) correspond to the *data integration* process. In what follows, we present in more detail these two processes.

8.3.3 Schema integration

The process of aligning different data source schemata results with three main outcomes [DS15]:

1. *Mediated/Integrated/Global schema*, which represents a unified and reconciled view of the underlying data sources [Len02]. It captures the semantics of the considered user domain, including the complete set of concepts of the data sources and the relationships among them.

2. *Attribute matching*, which specifies which attributes of the data sources' schemata correspond to which concept of the global schema. This matching can be 1-1, but sometimes one concept of the global schema may correspond to the combination of several attributes in the data sources' schemata.

3. *Schema mappings* are then built based on the previously found attribute matchings and they specify the semantic relationships (i.e., correspondence) between the schema of a data source (i.e., *local schema*) and the previously established *global schema*, as well as the transformations needed to convert the data from local to global schema.

8.3.3.1 Schema mappings

Schema mappings are typically expressed as views (or queries) either over the data sources' schemata or over the global schema.

First, we distinguish 3 main kinds of schema mappings, i.e., *sound*, *complete*, and *exact* [FM18], depending on the data they intent to map ($q_{Desired}$) and the data returned by the query of the mapping ($q_{Obtained}$).

- *Sound mappings* ($q_{Obtained} \subseteq q_{Desired}$) define the correspondence in which the data returned by the mapping query is a *subset* of those required, but it may happen that some of the required data is not returned by the query.

- *Complete mappings* ($q_{Obtained} \supseteq q_{Desired}$) on the other hand, define a correspondence in which the data returned by the mapping query include all those data that are required, but it may happen that some other data are returned as well (i.e., the returned data are a *superset* of the required data).

- *Exact mappings* ($q_{Obtained} = q_{Desired}$) finally are combination of both *sound* and *complete* mappings, meaning that the mapping query returns exactly the data required by the mapping, no more and no less.

Furthermore, we can implement schema mappings by means of two main techniques, i.e., *Global As View* (GAV) and *Local As View* (LAV).

In *GAV*, we characterize each element of global schema in terms of a view (or query) over the local schemata. Such approach enables easier query answering over the global schema, by simply unfolding the global schema concepts in terms of the mapped data sources, which is absolutely equivalent to processing views in a centralized RDBMS. However, this technique can result too rigid, having that a change over a single data source schema may require the change of multiple (and in the extreme case all) schema mappings.

In the case of *LAV*, we characterize elements of the source schemata as views over the global schema. LAV mappings are intended to be used in the approaches where changes in data source schemata are more common having that changes in the local schema only require updating the subset of mappings for the affected data source. However, the higher flexibility of LAV mappings comes with the higher cost of answering the queries over the global schema, which implies the same logic as answering queries using materialized views, already known as a computationally complex task [Hal01].

There are also some research attempts to generalize the previous two techniques, like *Global/Local As View*, which maps a query over the global schema in terms of a query over the local schemata, thus benefiting from the expressive power of both GAV and LAV techniques [FLM+99], and *Peer To Peer (P2PDBMS)*, in which case each data source acts as an autonomous "peer", while separate (*peer-to-peer*) mappings are defined among each pair of data sources [DGLLR07].

Despite their rigidness, GAV is the most widely used schema mapping approach in production data integration systems because of its simplicity.

8.3.4 Data integration

Once the schemata of different data sources are aligned, we move to the instance level. Here, as we mentioned before, there are two main steps. First, we look for the matching of records of different data sources (i.e., *entity resolution*) and then, we perform merging of the previously matched records such that at the output we have a single record that represents the object in the real world.

8.3.4.1 Entity resolution

Entity resolution tackles the problem of finding whether two (or more) records (typically from different data sources) represent the same object in the real world [GUW09].

For instance, if the records are representing individuals with their `name`, `address`, and `phone number`, to find the records that represent the same individual, it is not sufficient to only look for exact/identical values of these records, due to various factors that may affect how these values are represented: (1) *misspellings*, which may occur due to similar pronunciation ("Smythe" vs "Smith") or adjacency of letters on the keyboard ("Jones" vs "Jomes"); (2) *variant names* that may differ depending on the form in which the name is provided (e.g., with full first name and middle initial "Susan B. Williams", with initial of the first name "S. Williams", or with nickname "Sue Williams"); (3) *misunderstanding of foreign names*, that may occur in the cases in which one form may be assumed as default (e.g., Name Surname), while in other countries the typical order may be reverse, so "Chen Li" may (or may not) refer to the same individual as "Li Chen"; (4) *evolution of values* that occur when two records that represent the same object are created at different point in times, so some fields (e.g., phone number, address, marital status) may differ; and (5) *abbreviations* that may cause that the same objects (e.g., address) are spelled differently in different data source (e.g., "Sesame Street" vs "Sesame St.").

Therefore, in the entity resolution process, we need to look carefully at the kinds of discrepancies that occur and devise a scoring system or other tests that measure the similarity of records. Ultimately, we must turn the score into a yes/no decision in order to answer the question whether the records represent the same entity or not.

Some quick simplifications may be used to tackle the problem of value discrepancy and make the further matching process easier, like *normalization* of the strings (e.g., using first name initial without middle initials, or using abbreviation dictionary to always replace abbreviations with their full names). Also, we can decompose otherwise complex problem of entity resolution of the entire records, and do the comparisons field by field.

Similarity function ($\approx$). In a general case, in order to find how "close" are two values, we need to employ an effective *similarity function*. Depending on the expected level of discrepancy, and the type of values, we can use different methods to measure their similarity:

- A simple *Hamming distance*, which for two strings of the same length, measures the number of positions at which the corresponding symbols are different (e.g., $Hamming(\text{"}Jones\text{"},\text{"}Jomes\text{"}) = 1$, $Hamming(\text{"}Indiana\ Jones\text{"},\text{"}Indyana\ Jomes\text{"}) = 2$), or
- its more generalized version that works for any two strings, called *Edit distance*, which measures the minimum number of operations (*additions*, *deletions*, *substitutions*) needed to convert one string value to the another (a.k.a. *Levenshtein distance*).

However, these measures return an absolute "distance" between two strings, which may not always be reliable to decide whether two string represent the same object or not (e.g., $Levenshtein(\text{"}UK\text{'},\text{"}US\text{"}) = 1$ may be much more significant distance than $Levenshtein(\text{"}Unyted\ Kinqdon\text{'},\text{"}United\ Kingdom\text{"}) = 3$). To address this issue,

- *Jaccard similarity* represents another option for the similarity function, and it measures the similarity of two sets (where strings can be seen as sets of characters) relative to the their total size (i.e., $\frac{A \cap B}{A \cup B}$). Going back the example above, we can see that *Jaccard*("UK','US")=0.33 is comparatively smaller than *Jaccard*("Unyted Kinqdon','United Kingdom")=0.67, hence in this case Jaccard similarity is more effective as the similarity function.

Finally, regardless of the method used, the resulting similarity value must be converted to a binary (yes/no) decision, which is typically done by establishing a case-specific threshold (e.g., we can consider that if $Jaccard(A, B) \geq 0.5$, then $A \approx B$).

8.3.4.2 Record merge

Once the two records are found to be similar, the next step is to merge and replace them by a single record, that would further represent the real world object. Therefore, we need to employ an effective **merge function ($\wedge$)**.

For instance, we may take the *union* of all the values, or we may somehow *combine the values* to make a single one, in which case the approach would depend on the type of the values (e.g., we may establish a rule that a full name should replace a nickname or initials, and a middle initial should be used in place of no middle initial - hence "Susan Williams" and "S. B. Williams" would be combined into "Susan B. Williams"). However, the rule to be used may not always be obvious, especially in the case of misspellings.

The problem becomes even more complex when we consider the complete records, where different fields may use different similarity and merge functions. For instance, if two "similar" records disagree in one field, to merge these records we could choose to *generate null* values for that field, to take the *union of all the values*, or to choose the value depending on the *trustworthiness* of the data source from where the record is coming. Alternatively, we can also use users' feedbacks to establish the more accurate value (i.e., by means of *crowdsourcing*) [DS15].

8.3.4.3 R-Swoosh algorithm

Lastly, once the *similarity* and *merge* functions are decided and they satisfy certain set of properties[1], we can execute a simple algorithm (*R-Swoosh*, see Listing 8.2) to iteratively merge all the possible records [GUW09].

```
O:= ∅;
while (I<>∅) do {
        pick up r∈I;
        if (∃s∈O so that s≈r){
```

[1] Idempotence, Commutativity, Associativity, Representatibity (ICAR) properties. See [GUW09](21.7.3) for more details.

```
                I:=I-r; O:=O-s; I:=I∪{r∧s};
        } else {
                I:=I-r; O:=O∪{r};
        }
}
```

Listing 8.2: R-Swoosh algorithm

The R-Swoosh algorithm considers at the input a set of records from the underlying data sources (I) and starts with an empty output set O. In each iteration, it takes out one record r from I and it looks for the "similar" record s in the output. If matching is found, r is removed from the I and s from O, and the "merging" of these two records is put back to I so that the matching and merging with other records can continue. Otherwise, the record does not have matching with any record so far and thus it is removed from I and added to the output. Notice that in such way it is guaranteed that the previously merged records will be potentially matched and merged with all the other records from the input. Finally, the algorithm returns a set of all "uniquely" merged records (i.e., those that cannot be further merged with any other records in the output).

Multimedia Materials

Problem Definition (en)

Problem Definition (ca)

Kinds of Heterogeneity (en)

Kinds of Heterogeneity (ca)

Overcoming Heterogeneity (en)

Overcoming Heterogeneity (ca)

Schema Integration (en)

Schema Integration (ca)

Data Integration (en)

Data Integration (ca)

Chapter 9

Spatio-temporal Databases / Data Warehouses

Most of human and organizational behavior has some indication of time and space. It is estimated that up to 80% of data stored in databases include location or spatial components (see [VZ14]). Time as well represents an important aspect of day-to-day phenomena (e.g., employee's salary can change over time, time history of patient diagnosis, disease prevalence can change over time at a certain territory).

All this represents an opportunity to acquire space/time-oriented knowledge about our data. For instance, to verify that a territory is free of some infectious disease, we need to monitor the trends of number of new cases across the territory (e.g., counties, provinces, cities, and villages within the country), over time (e.g., in the last three years). In addition, data values as well can have attached the notion of the *time-validity* (e.g., customer can have certain home address in one period of time, and another address in a different period of time, and both should coexist in the same database).

To enable such data analysis, we need effective and efficient methods for representing and processing spatial and temporal information. Conventional databases typically use alphanumerics to describe the geographical information (e.g., textual names of city/province/country, or integer value for their respective populations), and they can only represent a state of an organization at a current moment, without further notion of the time dimension.

In this chapter, we introduce the main concepts of temporal and spatial support in RDBMS, and the extension to model a spatio-temporal DW.

9.1 Temporal databases

Many data values in our databases can vary over time (e.g., stock prices can grow or drop on a daily bases, patient medical status can change over the period of time, flight status can go from taxing - taking-off - in-flight - landing), making it important to support the analysis of the data through the time dimension (i.e., over time to follow the trends, or at a specific moment to capture a snapshot).

9.1.1 Theoretical foundations of temporal DBs

Incorporating temporal information in the data management (by means of attaching time periods to traditional DB columns), allowed storing and analyzing database states at different moments in time. Database can support two main dimensions of time, i.e., *transaction time* (which is *implicitly logged* in the DBMS) and *valid time* (which needs to be *explicitly stated* as a temporal attribute). In addition, there is also the notion of *user-defined time*, that corresponds to specific semantics and knowledge to the database user (e.g., *birthday*, *hiring date*). Managing this temporal dimension is not directly supported by DBMS (like transaction and valid time), but it rather needs to be interpreted for its specific semantics at the application level.

9.1.1.1 Transaction time support

Transaction time serves to indicate the evolution of time in terms of occurred database transactions. This way we can model the database reality (and history) in terms of tuple insertions, updates, and deletions. Hence, database records are extended to include a time interval $[t_1, t_2]$, and this interval is modified as follows in the moment of committing transactions:

- **On insert** at time t_1, a right-open time interval $[t_1, now)$ is added to the record, indicating that the inserted record is currently in use.
- **On update** at time t_2, the time interval of the current record is updated to $[t_1, t_2]$ (i.e., closed), and a replica of the record is created with the right-open time interval $[t_2, now)$.
- **On delete** at time t_3, the record with the open time interval is updated to $[t_2, t_3]$ (i.e., closed), indicating that the record is not in use anymore for DML operations, but the record is not physically removed. Instead, it remains stored for keeping the complete history of record values.

By storing the complete history of records' values in the table, we can reproduce a database table state at any point of time in the past, while only the records with currently "open" interval are used in the present.

9.1.1.2 Valid time support

Contrary to the transaction time, *valid time* models the temporal validity of the real world phenomena related to database values (i.e., the period of time in which a data value is true in the modeled reality). For example, validity period of a contract, validity of credit card, user subscriptions to a service (which is independent of when the contract, credit card or subscription is actually registered in the database).

Valid time support needs to be explicitly defined for the column in our database table, which then adds two timestamp fields to that column (*start* and *end* of time validity). To define the validity of a column value, the user needs to explicitly assign start (and end) timestamps. This cannot be automated by the system like in the case of transaction time. Also unlike transaction time, the valid time periods can be in the past, present, and even in the future. This further allows to query, update, delete, or constrain the database tables using a *temporal validity component*. For example, we can query only the employees that are *currently* working for a department, or the employees that worked for a department *at all times*, but as well the employees that will work *until a specific time point* in the future (e.g., next 2 years). See [SBJS96] for more details of this.

9.1.1.3 Bi-temporal support

Some DBMS (like Oracle from v12c) are completely "time-aware", meaning that they provide support for both *transaction* and *valid times* (i.e., *bi-temporal support*). DBMSs with bi-temporal support include additional columns for both valid and transaction time, while these times are separately when performing DML operations.

For example, if a row in a bi-temporal table is deleted, the ending bound of the transaction-time period is automatically changed to reflect the time of the deletion, and the row is closed to further DML modifications. The database reality, reflected by the modified ending bound of the transaction-time period, is that the row has been deleted. The valid-time period of the row however remains unchanged, because the deletion does not affect the ending bound of the valid-time period, the row information retains its character in the valid-time dimension. Nevertheless, because the row was deleted, the row does not participate in further DML operations for the table, even though it remains in the table as a closed row in transaction time.

This allows to capture a snapshot of our database table from both valid and transaction point of view

9.1.2 Implementation considerations (Temporal DMBS)

In practice, besides the temporal data models (i.e., adding temporal information to existing attributes), the query language (SQL) is as well extended to support temporal query processing. The most known

SQL extension that supports temporal data aspects is SQL3 (see [SBJS96]), which collects and extends the temporal support previously proposed and accepted for TSQL2 (see [S+95]). Although SQL3 was never accepted as a standard, its temporal support components are included in the *SQL:2011 standard*, implemented in many of the most used commercial DBMS (e.g., Oracle, Teradata, IBM DB2, MS SQL Server). Such extension provides support for expressing temporal information about our data, and as well processing temporal queries (e.g., different types of temporal joins and temporal range queries).

Oracle has a bi-temporal support through which it can handle both transaction time (through *Flashback Data Archive* component) and valid time (through its *Temporal Validity* feature). *Flashback Data Archive* allows to monitor query trends, keep table history, navigate through past and recover database table in certain point in time. Such support is also important for leaving audit trails in the DBMS.

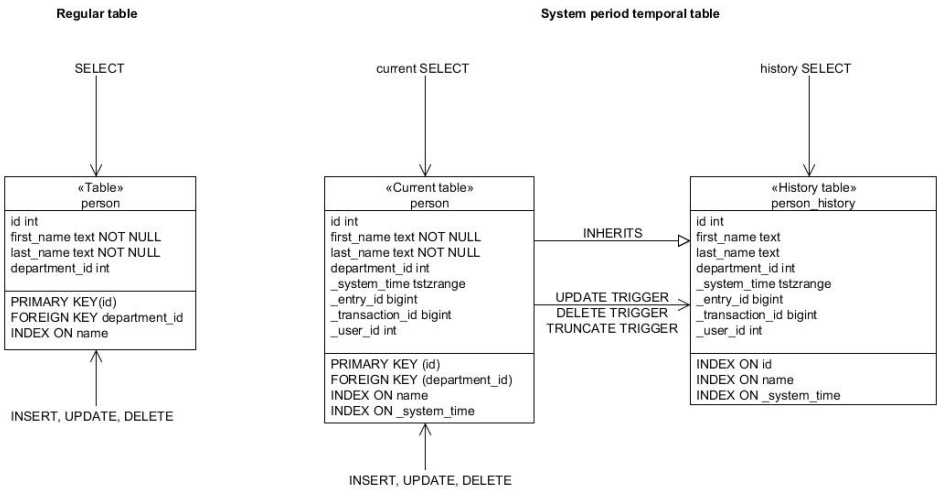


Figure 9.1: Example of System Period Temporal Tables in PostgreSQL

PostgreSQL has an extension that provides a custom-made transaction time support, which is however not fully compliant with SQL:2011.[1] The support is implemented through *System Period Temporal Tables*, which besides the *Current table* with records being in use also stores a *History table* where the historical versions of table records are stored. As expected, applying DML operations (*UPDATE*, *DELETE*, *TRUNCATE*[2]) over the tables with temporal support does not cause permanent changes to the table, but rather creates new versions of the data in the history table. To automatically support the corresponding actions at the moment of applying operations, several triggers are implemented and fired when the operations are applied. An example of *System Period Temporal Tables* system for the *person* table is showed in Figure 9.1. *Current table* has the same structure and name as original table, except for four technical columns (i.e., *_system_time* - time range specifying when this version is valid, *_entry_id* - id of entry that can be used to group versions of the same entry or replace PK, *_transaction_id* - id of transaction that created this version, and *_user_id* - id of user that executed modifying operation that created the entry). Values for these columns are always generated by the system and the user cannot set them. To ensure backward compatibility, these columns can be marked as implicitly hidden. *History table* also has the same structure and indexes as current table, but it does not have any constraints. History tables are insert only and the creator should prevent other users from executing updates or deletes by defining the appropriate user rights. Otherwise the history can result inconsistent. Lastly, *triggers* for storing old versions of rows to history table are inspired by referential integrity triggers, and are fired for each row after UPDATE and DELETE, and before each TRUNCATE statement.

[1] `https://wiki.postgresql.org/wiki/SQL2011Temporal`

[2] *TRUNCATE* operation deletes all records in a table (like an unqualified DELETE) but much faster.

9.2 Spatial DBMSs

To represent spatial data in databases, in addition to the traditional data types (e.g., text, integer, float), three main components need to be included in the DBMS.

1. **Reference system.** A coordinate-based system used to uniquely define geometrical objects and their position (projection) within the given system.

2. **Spatial data types.** Extension of database types to store geometry data (e.g., points, lines, polygons).

3. **Spatial operations.** Extension of the database operation set to manipulate previously defined spatial data types (e.g., check if one polygon contains or overlaps with another polygon, line or point, union/intersection/difference of polygons).

Apart from this, given the complexity of spatial data types, traditional access methods need to be extended to allow efficient querying and processing of spatial data - **spatial indexes**.

9.2.1 Reference system

A spatial reference system (SRS) assigns coordinates in a mathematical space to a location in real-world space [VZ14].

The most commonly used reference system to represent spatial data is a geographic coordinate system which associates geometries to their positions on the planet Earth, also known as *ellipsoid*. Ellipsoid is a mathematical approximation of the Earth's surface, defined by the World Geodetic System in 1984 and last revised in 2004. It is as well used by the Global Positioning System (GPS).

The ellipsoid is used to measure the location of points of interest using *latitude* and *longitude*. These are measures of the angles (in degrees) from the center of the Earth to a point on the Earth's surface. Latitude measures angles in the North–South direction, while longitude measures angles in the East–West direction. Given that ellipsoid SRS is an approximation, it needs to be defined separately for each specific region of the globe. If two geometries are in the same SRS, they can be overlaid without distortion, otherwise, they need to be transformed.

To produce a map, the curved surface of the Earth, is approximated by an ellipsoid and transformed into the flat plane of the map by means of a map projection. Thus, a point on the reference surface of the Earth with geographic coordinates expressed by latitude and longitude is transformed into Cartesian (or map) coordinates (x, y) representing positions on the map plane.

9.2.2 Spatial data types

There are two main approaches to represent spatial data inside a database, i.e., **object-based** and **field-based**. The *object-based* approach is primarily used to represents the identifiable objects with their projection to the reference system and their shapes (e.g., road as a line, country boundaries as a polygon). The *field-based* approach sees the world as a continuous surface over which different phenomena may vary. It thus then allow to associate with each point a value that characterizes a certain feature at that point (e.g., temperature, altitude), see [VZ14].

Spatial data types are adopted by RDBMS using the *OGC standard*.[3] OGC standard defines spatial data types through a taxonomy of geometry types, with *Geometry* as a root, all projected to the reference coordinate system (i.e., *SpatialReferenceSystem*). See Figure 9.2 for more details, and notice that we focus here on the shaded classes. In particular, the main distinction is made between:

- *Base geometries:*
  - *Point*, which represents a single point in the reference coordinating system, defined by a pair of decimal values, i.e., longitude and latitude (e.g., town, health facility, airport).

[3] https://www.ogc.org/standards/sfas

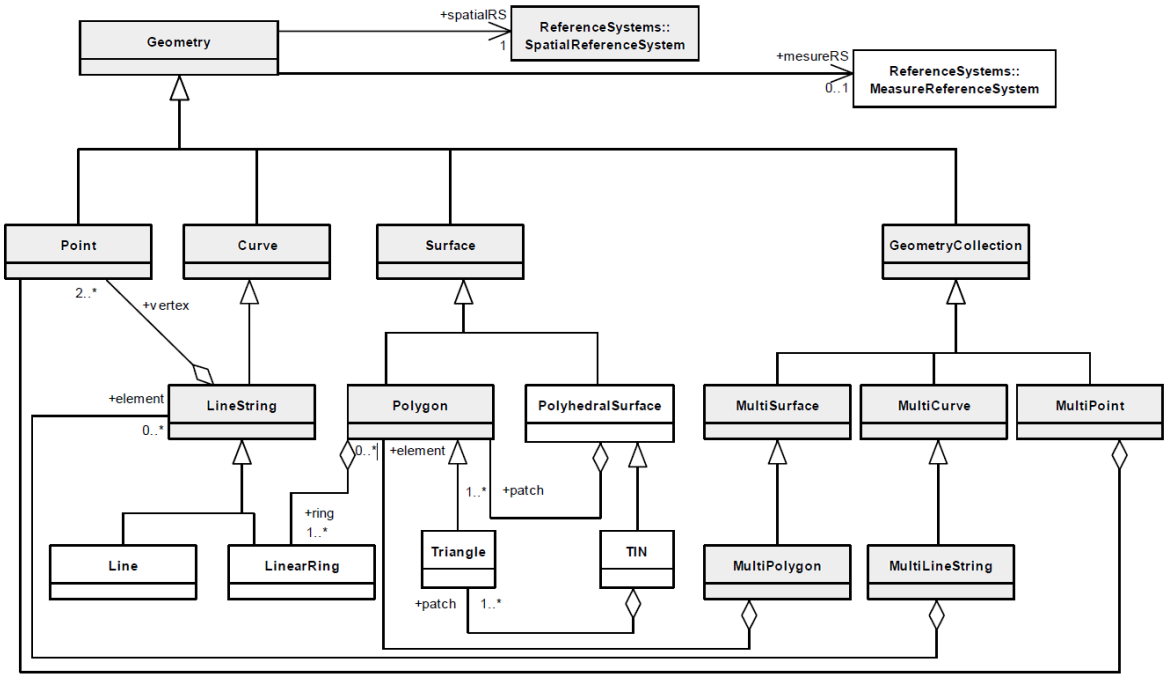


Figure 9.2: Geometry taxonomy of spatial data types*

* https://portal.ogc.org/files/?artifact_id=25355

  - *Curve* (implemented as *LineString*), which represents a one-dimensional object stored as a sequence of points defining a linear (open) path (e.g., railroad, highway, tracking path).
  - *Surface* (implemented as *Polygon*), which represents a two-dimensional geometric object stored as a sequence of points defining its exterior (closed) boundary and zero or more interior boundaries (e.g., province, county, lake).

- *Homogeneous collections of geometries:*

  - *MultiPoint*, which represents a 0-dimensional geometry collection, where point elements of the collection are not connected nor ordered in any semantically important way (e.g., set of province capitals in a country, set of health facilities in a city).
  - *MultiCurve* (implemented as *MultiLineString*), which represents a collection of curve geometries (e.g., country's railroad system, city's metro lines).
  - *MultiSurface* (implemented as *MultiPolygon*), which represents a collection of surface geometries (e.g., country's territory with exclaves and islands, archipelago of islands).

9.2.3 Spatial operations

To operate over the previously presented spatial data types, a set of spatial operations are introduced next.

The main kinds of spatial operations are as follows (see [VZ14]):

- **Numeric**, which receive as input one or more geometries and return a numeric value (e.g., length, area, perimeter, distance, direction).

- **Predicates**, which receive as input one or more geometries and return a boolean value (e.g., IsEmpty, OnBorder, InInterior).

- **Unary operations**, which receive as input one geometry and return new geometry resulting from the applied transformation (e.g., Boundary, Buffer, Centroid, ConvexHull).
- **Binary operations**, which receive as input two geometries and return new geometry resulting from the applied transformation (e.g., Intersection, Union, Difference, SymDifference, Spatial join).

A special type of spatial predicate operations are those that indicate topological relationship that exist among input geometries, i.e., **topological operations** (see some self-explanatory examples in Figure 9.3). Notice that *Contains/Within* is a specific version of the *Covers/CoveredBy* operation. Contains requires that one geometry is located in the *interior* of the other geometry and hence that the boundaries of the two geometries do not intersect, while Covers does not pose such restriction and includes also the cases where the boundaries of two geometries may intersect.

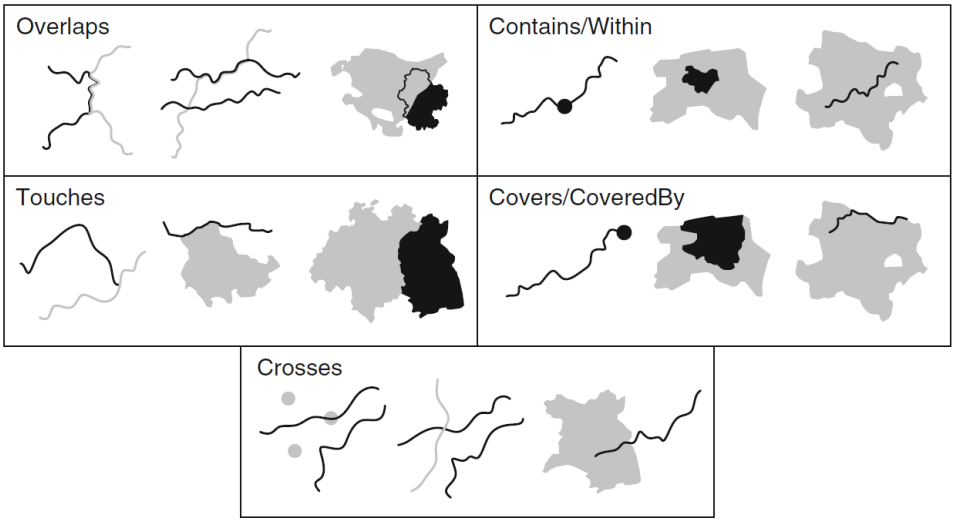


Figure 9.3: Examples of topological relationships between two geometries (in gray and in black) [VZ14]

To compare any two geometries it is required to compare (for *intersection*) their *exterior* (i.e., all of the space not occupied by a geometry), *interior* (i.e., space occupied by geometry), and *boundary* (i.e., interface between a geometry's interior and exterior). In particular, an intersection between two geometries can result in: *point* (0), *curve* (1), *surface* (2), or *no intersection* (-1). For each topological operation, it is then needed to define one or several *pattern matrix*[4] (analogous to a regular expression), over which the results of intersections among their interiors, exteriors, and boundaries are matched. A pattern matrix can have the following cells: *T* - there is intersection of any type (point - 0, curve - 1, or surface - 2); *F* - there is no intersection (-1); * - intersection is irrelevant (-1, 0, 1, or 2); and the intersection must be of a specific type (i.e., 0, 1, 2).

An example of a pattern matrix for the topological operation *Contains* is presented in Figure 9.4. It is obvious to see from Figure 9.3 that in order for a geometry *a* to *contain* geometry *b*, there must be an intersection between their interiors (of any kind, depending on the type of geometry), while the exterior of *a* must not intersect with the interior nor with the boundary of *b* (having that *b* must be entirely contained within *a*).

9.2.4 Spatial indexing techniques

To efficiently support spatial operations and querying (e.g., searching for overlaps, nearest neighbor or doing spatial joins), analogous to the traditional RDBMS, we need to employ access methods (i.e., indexes). However, traditional indexing techniques (B^+-trees or hashing) are failing here given that spatial data typically lack total ordering (see [MTT18]).

[4]Examples of pattern matrices for all topological operators can be found at: `https://desktop.arcgis.com/en/arcmap/10.4/manage-data/using-sql-with-gdbs/relational-functions-for-st-geometry.htm`

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	*
	Boundary	*	*	*
	Exterior	F	F	*

Figure 9.4: Pattern matrix for the *Contains* operation

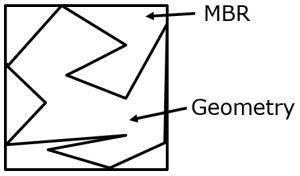


Figure 9.5: Example of Minimum Bounding Rectangle

The main challenge when it comes to processing geometries is due to the complexity of their representations. For example, a MultiPolygon that is used to represent Norway with all its fjords and islands can contain very large quantity of points to precisely describe the shape, thus processing it can be a very CPU-intensive task. To this end, when building indexing structures (i.e., trees), approximations of these shapes are often used. One of the typically used approximations is Minimum Bounding Rectangles (MBR), with the sides of the rectangle parallel to the axes of the data space (an example of an MBR is depicted in Figure 9.5). Notice that we further rely on such approximations to be stored inside the index structure, while the real geometry is stored separately in the disk.

We classify spatial indexing techniques based on the two main characteristics:

- **Space that the index is covering.** Here we distinguish between *space-drive structures*, which partition the entire space under consideration and map indexed objects to that space, and *data-driven structures*, which partition only the part of the space containing the indexed spatial objects.
- **Objects that are indexed.** Here we distinguish between *point access methods*, a.k.a. *multidimensional points*, which can index points inside the considered space, and *spatial access methods*, a.k.a. *multidimensional regions*, which can index more complex spatial types, like surfaces/polygons.

Here, we will present two representative examples of spatial indexes, **R-tree** and **Quadtree**.

9.2.4.1 R-tree

R-tree is a representative of *data-driven structure* and *spatial access method*, which is the base of most famous commercial DBMS implementations of spatial indexes. It is a height-balanced tree (based on B+-tree) that consists of (1) *leaf nodes* that consist of a collection of pairs [MBR of that geometry object, reference to the geometry object on the disk]; and (2) *intermediate nodes* that consist of collections of pairs [*MBR enclosing* all MBRs of the child node, reference to the child node in an R-tree]. An example of an R-tree and corresponding indexed geometries is shown in Figure 9.6.

To perform filter or join queries over the indexed geometries, we first follow the MBRs from the root of the R-tree (from coarser to finer level) until the leaf level is reached. Similarly, to insert a new geometry, from the root of the R-tree, we follow the child node *that needs minimum enlargements* to fit geometry's MBR, until the leaf is reached. If the leaf is overfilled, the node is *split* and the entries are redistributed over the current and new nodes. The main task of node split algorithms is to find a "proper" distribution of all geometries over the two nodes after splitting. They typically also follow the minimum enlargements criteria but they can become very complex in general. The original exhaustive algorithm and some alternative approximations are given in [HHT05].

9.2.4.2 Quadtree

Quadtree is a representative of *space-driven structure* and *spatial access method*, typically used for representing binary (black & white) images, represented as $2^n x 2^n$ matrix in which 1 stands for black and 0 for white. The index structure is based on degree four tree of max height n. Having that it is a space-driven

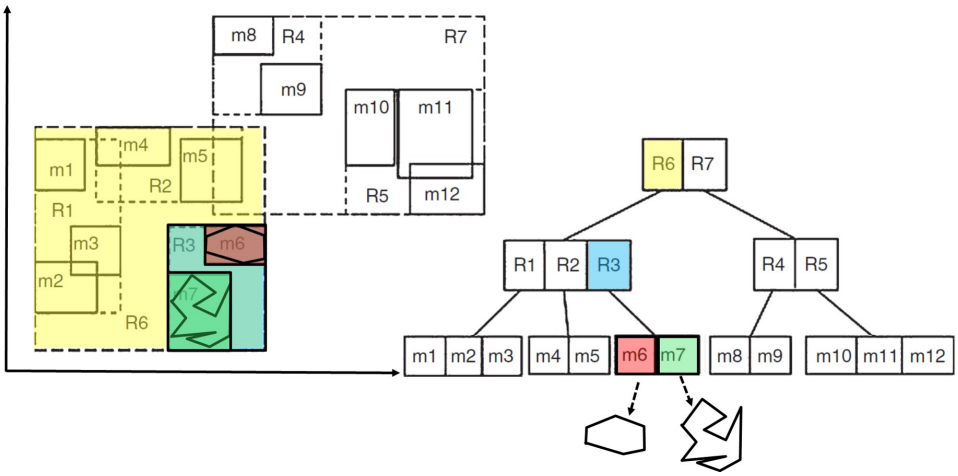


Figure 9.6: Example of R-tree (inspired by [MTT18])

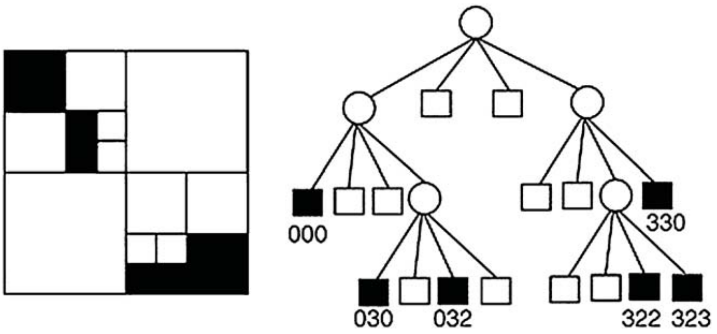


Figure 9.7: Example of Quadtree [MTT18]

structure, the root of the tree corresponds to the whole image (the entire space). Each tree node then corresponds to an array of four equivalent image parts (from upper left to lower right) and depending on the part "color filling" can be: (1) *intermediate*, represented as a circle, meaning that the part has multiple colors; and (2) *leaf*, represented as a black or white square, meaning that the part is completely colored either in black or in white, respectively (see an example of Quadtree in Figure 9.7).

9.3 Spatial Data Warehouses

Traditional DW models typically include a geographic dimension (e.g., *Airport*→City→Country), but without a real spatial support for the analysis (i.e., no spatial relationship between dimension levels nor spatial measures).

To conceptually model spatial DW, [PSZ06] introduced MADS spatio-temporal model, corresponding to the OGC standard geometry taxonomy (see Figure 9.8), and extended with the icons that illustrate the nature of these geometric objects. Besides these data types, MADS model also defines a set of topological operations (see Figure 9.9), that may exist among dimensional concepts, with their corresponding icons. Using such spatial extensions, we further see how spatial dimensions (and their hierarchies) and spatial measures can be defined.

9.3.1 Spatial hierarchies

The MADS conceptual model allows us to enrich traditional geographical dimensions with a specific data type of each of the dimension levels (e.g., Airport as *Point*, City as *Surface*, and Country as *Mul-*

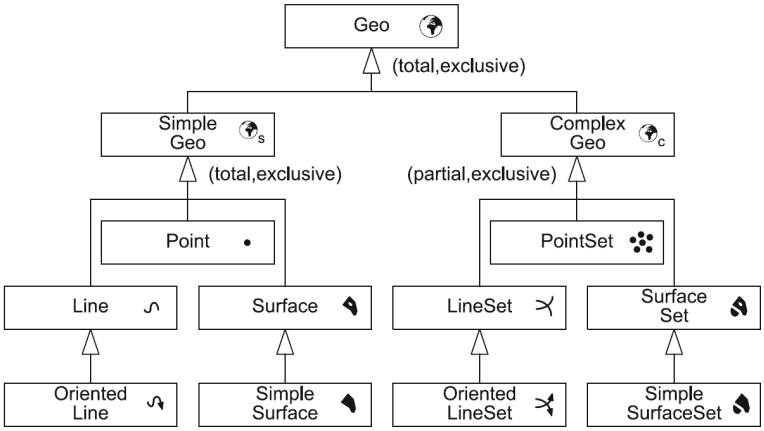


Figure 9.8: MADS spatio-temporal model (spatial data types icons) [PSZ06]

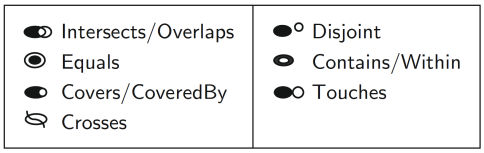


Figure 9.9: MADS spatio-temporal model (topological operations icons) [PSZ06]

tiSurface; see Figure 9.10), as well as the topological relationship that exists between the dimensional levels (e.g., Airport is *CoveredBy* City, which is *CoveredBy* Country). Notice that this enriches a traditional dimension, typically based on the multiplicities of the levels in the level-to-level relationship (e.g., Airport belongs to one and only one City), with spatial specific semantics (e.g., Airport is *CoveredBy* by a City, City is *Within* a Country).

9.3.2 Spatial measures

Similarly, besides traditional numerical measures, MADS allows for defining spatial measures in our fact table. Besides simply adding a spatial icon to indicate that a measure is spatial (e.g., FlightArea as *Surface*; see Figure 9.11), we also need to define a special spatial aggregation function, used to evaluate a spatial measure at different levels of granularity.

In particular, analogous to traditional aggregation functions we distinguish between three main kinds of spatial aggregation:

- **Distributive**, those that can be computed in a distributed manner, i.e., partitioning data and applying aggregated function over smaller subsets, and then applying the same function over the results of each partition. In traditional DW, examples of distributive aggregation functions are *sum()*, *count()*, *min/max()*. In the case of spatial DW, examples of distributive spatial functions are: *convexHull()* (the smallest convex polygon, that encloses all of the geometries in the input set), *spatialUnion()*, *spatialIntersection()*.

- **Algebraic**, those that can be computed by an algebraic function with N arguments, each of which is obtained by applying a distributive aggregation function. In traditional DW, an example of algebraic aggregation function is *avg()*, which can be computed by *sum()/count()*. In the case of spatial DW, an example of algebraic spatial function is center of n points (*centroid()*). For a set of n points (i.e., $S = \{(x_1, y_1), (x_2, y_2), ...(x_n, y_n)\}$), we compute a *centroid(S)* as $(\frac{1}{n}\sum_{i=0}^{n} x_i, \frac{1}{n}\sum_{i=0}^{n} y_i)$.

- **Holistic**, in the case that there is no algebraic function that characterize the computation, meaning that there is no way to distribute the computation the measure value. In traditional DW, an example of holistic aggregation function is *median()*. In the case of spatial DW, an example

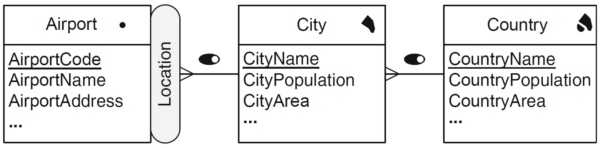


Figure 9.10: Spatial hierarchy example (inspired by [VZ14])

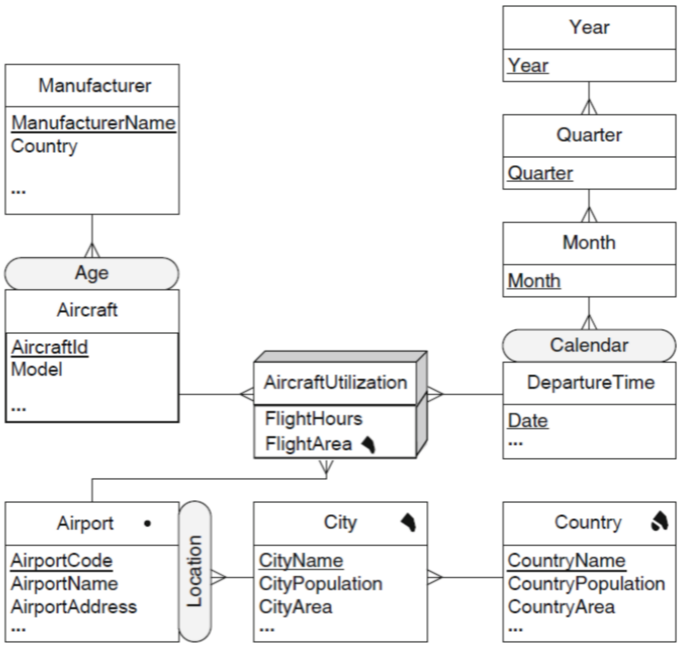


Figure 9.11: Spatial measure example (inspired by [VZ14])

of holistic aggregation function is *equipartition()*, which partitions convex bodies into equal-area pieces, creating the *equipartition*.

Multimedia Materials

Temporal Databases (en)

Spatial Data Types (en)

Spatial Operators (en)

Conceptual Design of Spatial DW (en)

Spatial Indexing Techniques (en)

Chapter 10

Dashboarding

With contributions of Daniele Perfetti

Data visualization can be seen as a use of computers to better comprehend existing data or to extract additional (not easily seen) knowledge from the data that resulted from simulations, computations, or measurements [McC87].

I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

(a) Data sets

Property	Value
Mean of x	9
Sample variance of x	11
Mean of y	7.50
Sample variance of y	4.125
Correlation between x and y	0.816
Linear regression line	$y = 3.00 + 0.500x$
Coefficient of determination of the linear regression	0.67

(b) Statistic properties

Figure 10.1: Anscombe's quartet data

For instance, the importance of graphical representation of data can be seen in a famous example of Anscombe's quartet[1]. In Figure 10.1a, we can see four different datasets of Anscombe's quartet. Just by analyzing their (almost exact) simple summary statistics (see Figure 10.1b), we could conclude that these four datasets are identical.

However, only after plotting the datasets in Figure 10.2, we can see how they vary considerably in the distribution. More specifically, the first chart (top left) shows a simple linear relationship, while the second one (top right) is obviously not linear, thus requiring more general regression techniques. The third chart (bottom left) is also linear, but its regression line is offset by the one outlier lowering the correlation coefficient. Lastly, the fourth chart shows how the one high point produces a high correlation coefficient while the other points do not indicate any relationship between the x and y variables.

This is only exacerbated in the case of Big Data, were these cannot be fully comprenhended by the limited human senses and brain. Indeed, human eye can only sense few megabytes per second, but it is even too much for the poorer brain learning capacity which is in the order of few tens of bits per

[1] https://en.wikipedia.org/wiki/Anscombe%27s_quartet

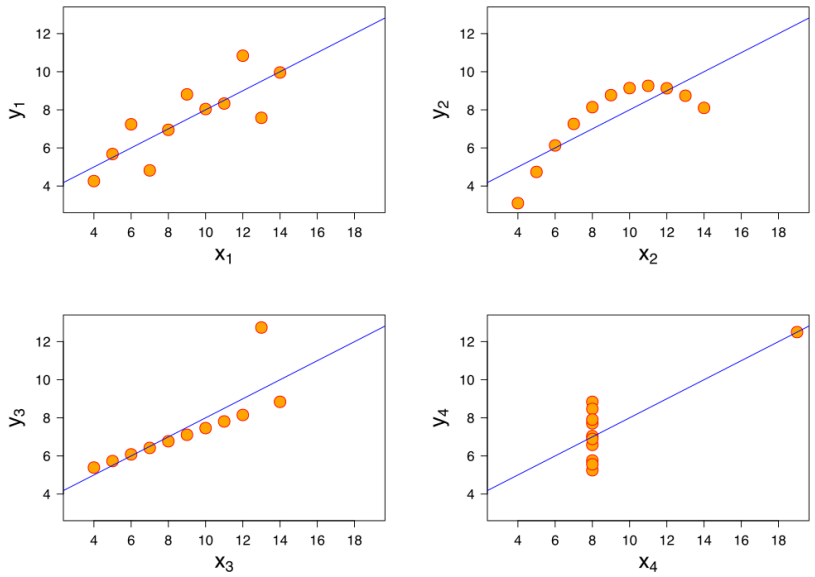


Figure 10.2: Anscombe's quartet - visualization

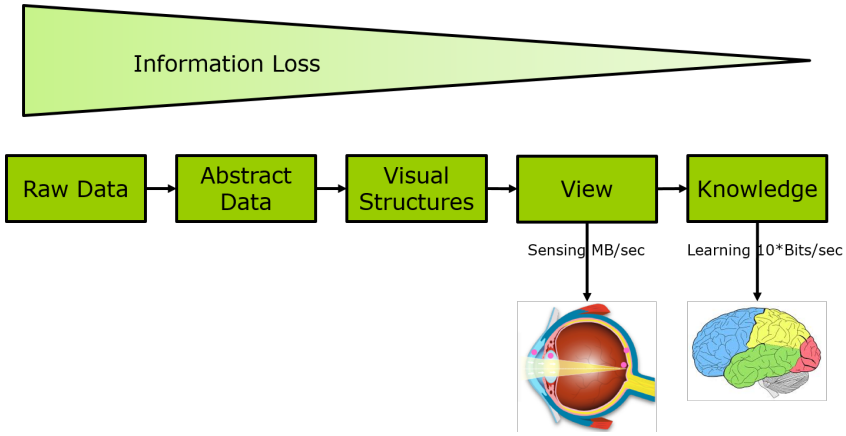


Figure 10.3: From data to insights

second. Therefore, as sketched in Figure 10.3, we must assume some information loss in the process of transforming raw data into knowledge that can be really consumed by our brain. Firstly, we have to abstract the data in the form of relevant features with the appropriate units and scales, and then structure them in the form of meaningful charts, so they convey the appropriate message.

Thus, we can identify the main three benefits of visualizing the data, namely to: (1) facilitate understanding of the data by providing simpler way to get an insight or information from large datasets, (2) facilitate pattern identification in the data, like outlier detection or identifying trends to decide on the next best steps, and (3) attract attention of the audiance by using appealing graphical images to communicate the data insights.

10.1 Dashboard definition

Stephen Few defines a dashboard as a "visual display for the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored and understood at a glance" [McC87].

Dashboards have easily become one of the most popular visualization tools in business intelligence and data warehousing, as they allow organizations to effectively measure, monitor, and manage business performance [VZ14].

Typically, dashboards consist of three main elements, namely, (a) Diagram(s) visualizing the measured metrics, (b) Heading explaining the content of the dashboard and its purpose, and (c) Short explanation (or interpretation) of the status and information in the diagram.

The following general characteristics are also recommended for an effective dashboard, namely: (a) that it fits in a single computer screen (thus avoiding scrolling or paging to find the right information), (b) that it contains minimal (necessary) information (so that it enables fast grasping of the needed information and avoiding unnecessary visual distractions), (c) that it shows the most important business performance measures to be monitored, (d) that it updates automatically from the data (daily, weekly, or when the data changes if more timely information is needed) in order to display up to date information, and (e) that it allows interaction with the displayed data for their better understanding (e.g., enabling filtering or drilling-down).

There are three main high-level classes of dashboards, depending on their purpose [VZ14]:

- **Strategic** dashboards, which provide a quick overview of the status of an organization for making executive decision (long-term goals). The focus of strategic dashboards is not on what is going on right now (no real-time data), but on the recent past performance.
- **Operational** dashboards are used to monitor organization's operations, thus requiring more timely data to track changing activities that could require immediate attention. Operational dashboards are recommended to be simple to enable rapid visual identification of monitored measures and quick reaction for correcting them.
- **Analytical** dashboards should support interaction with the data like drilling down into the underlying details, in order to enable exploration to make sense of the data. They should allow not only to examine what is going on, but also to examine the causes.

10.2 Key Performance Indicators (KPIs)

Key Performance Indicators (KPIs) represent complex measurements that are used to evaluate or estimate the effectiveness of an organization and to monitor the performance of their processes and business strategies in the dashboard [VZ14]. KPIs need to be aligned with the specific business strategy and defined following the main business objectives. These objectives should follow the *DUMB* properties,[2] that is, they need to be: (a) **D**oable (i.e., feasible to be achieved), (b) **U**nderstandable (i.e., could be interpreted by those who follows them), (c) **M**anageable (i.e., could be calculated or estimated in a tractable manner from the available data), and (d) **B**eneficial (i.e., useful to understand the status of an organization and making strategic decisions).

KPIs can be classified in various manners (see [VZ14] - Section 9.2.1 for more details), while one of the simplest generic classifications is based on their temporal aspect, namely: (a) **Leading KPIs**, which reflect the expectation of what is going to happen in the future (e.g., *expected demand*), (b) **Coincident KPIs**, which reflect what is currently happening (e.g., *number of current orders*), and **Lagging KPIs**, which reflect what happened in the past (e.g., *revenue before taxes* or *customer satisfaction*). These clearly relates to the three classes of dashboards presented in the previous section.

KPIs are most typically represented with a quantitative measure that can be evaluated from the available data, but there also may exist qualitative KPIs that measure a descriptive characteristic from the data, like opinion or a property.

We often compare KPIs with respect to a specific **target**, that is, a pre-determined value of the KPIs that indicates success or failure. In that case, KPIs can be monitored with a reference to how close/far they are from the defined target value, thus indicating to which extend a certain goal has been achieved.

Finally, to achieve our business objectives, we need to set more specific strategies (i.e., goals), and correspondingly define the related KPIs. These goals (and hence the KPIs) should preferably fulfill the

[2] https://www.kaushik.net/avinash/web-analytics-101-definitions-goals-metrics-kpis-dimensions-targets/#kpi

so-called *SMART* properties,[3] that is, they should be: (a) **S**pecific (i.e., more detailed than a related objective), (b) **M**easurable (i.e., could be measured and obtained from the data), (c) **A**chievable (i.e., could be attained - similar to the Doable property of the objective), (d) **R**elevant (i.e., realistic, related to the real business), and (e) **T**imely (i.e., time-bounded, with a pre-determined deadline).

10.3 Complexity of visualizations

There are several sources of complexity when it comes to creating an effective dashboard. On the one hand, there are various parameters whose variation may lead to a large set of possible visualizations, like attributes/variables, aesthetics, encoding, and the subset of data to be considered. On the other hand, another source of complexity is also the calculation of the values to be displayed in the visualizations (e.g., KPIs), which can be computationally expensive. In such case, value approximations are often considered to lower such a burden.

10.4 Dashboarding guidelines

Following from the previous section, we must then carefully choose the visual elements and the possible interactions within our dashboard so that the information is properly and easily conveyed to the intended audience. There are several practical guidelines[4] that should be followed when creating a dashboard.

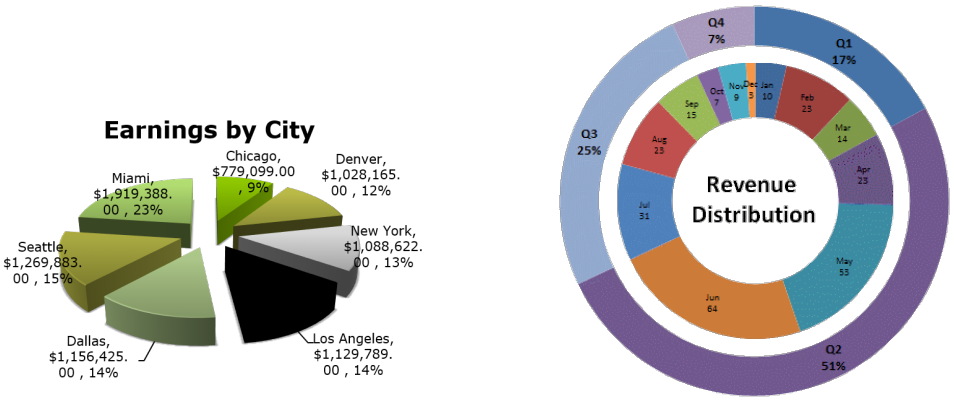


Figure 10.4: Unnecessarily complex chart design (H. A. Downs)

First of all, as mentioned before, the dashboard **should be kept simple**, containing only the necessary information, and using the proper amount of visual elements. The charts in the dashboard should not be difficult to read. If an end user has to think much to read a chart or ask how to interpret it, the design is probably not the most appropriate one. For instance, both charts in Figure 10.4 are overly complicated (too many visual elements) so that the necessary information (i.e., revenue distribution per months and quarters or earnings per city both in percentages and absolute values) can be grasped quickly. The objective of an effective dashboard is to transmit the necessary information, not to demonstrate how skilled the dashboard designer is.

However, the dashboards should **neither be overly simplistic** nor "boring". It is often recommended to use Gestalt principles[5] of proximity and similarity to include combination of aesthetics that can convey the information in visually more pleasing and understandable manner. For instance, in Figure 10.5,

[3] https://www.mindtools.com/pages/article/smart-goals.htm

[4] H.A. Downs. Focusing Digital Dashboards for Use: Considering Design, Planning and Context: https://comm.eval.org/researchtechnologyanddevelopmenteval/viewdocument/focusing-digital-dashboards-for-use

[5] https://en.wikipedia.org/wiki/Gestalt_psychology

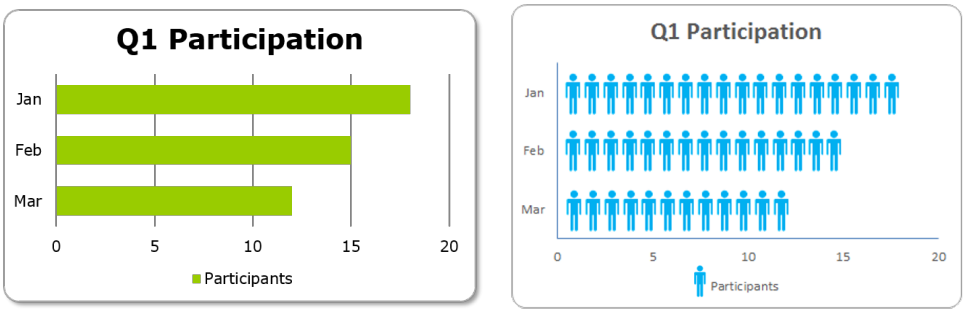


Figure 10.5: Overly simplistic chart design (H. A. Downs)

on the left, we can see a simple bar chart showing the number of participants per month. Although this chart is correct and simple enough to be understood, just by replacing basic bars by more intuitive aesthetics, i.e., the corresponding number of icons of participants, the chart (on the right) becomes much more attractive for the end user.

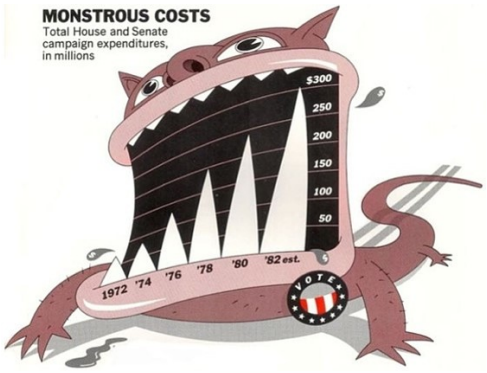


Figure 10.6: Example of aesthetics overuse in a chart (H. A. Downs)

While the use of such aesthetics can improve the attractiveness of our charts, they must be used with care since their overuse may cause the counter effect, and draw the attention of the end user to the design, instead of the information to be transmitted (see an example in Figure10.6).

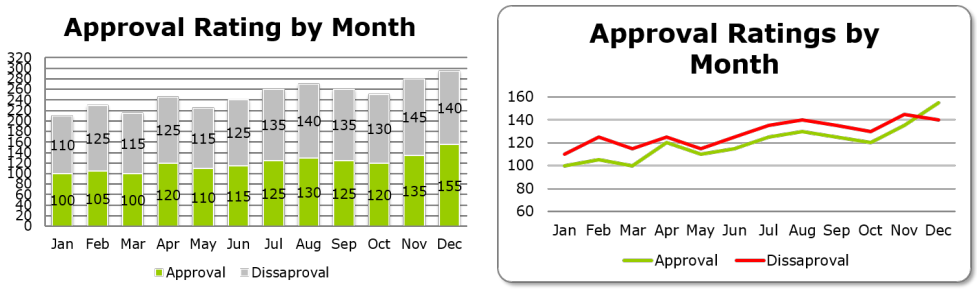
Finally, there are several specific guidelines or best practices suggested for achieving a "good chart", depicted in Figure 10.7.

Last by not least, choosing the appropriate type of chart to convey the desired information is as well an important step in the dashboard design. For instance, in Figure 10.8, two charts are showing approval ratings per month, including two groups (*approval* and *disapproval*), and both based on the same underlying dataset. Chart in Figure 10.8a is using stack graph, but with absolute values. Stack graph is actually more useful when set up as 100% stack graph, since it would much easier show the proportion among the two groups. However, as it can be seen in Figure 10.8b, line chart does a better job in showing between group differences. In particular, in the line chart we can easily see that the approval and disapproval ratings swap places in the last year, while in the stack chart this can be seen only by reading the numbers in the bars, which practically makes the visualization futile.

10.5 Design principles

Apart from the practical guidelines presented in the previous section, when designing charts in our dashboard, we should as well adhere to some general design principles (no limited to dashboards only) so that our charts are successful in transmitting the right information to its targeted audience. These

Figure 10.7: Anatomy of a good chart (H. A. Downs)



(a) Stack chart

(b) Line chart

Figure 10.8: Showing between group differences (stack vs. line chart) (H. A. Downs)

principles are based on laws, guidelines, human biases, and general design considerations, and are introduced in [BHL03], while here we discuss the ones that are most applicable to the dashboard design.

- **Chunking** refers to the technique where units of information are grouped into a limited number of chunks, making it easier to process and remember by human. This technique is related to the short-term memory usage that can accommodate up to four chunks at a time (+/- one). The simplest example is grouping the digits of a long phone number so that it can be observed and remembered more easily, where groups can as well have a specific purpose like country code, province code (e.g., 34934137889 vs. +34 93 413 7889). The same technique applies to the design of a web page containing a large amount of information, like in our case a dashboard containing many charts and their related metadata. Charts should be then grouped by topics (not more than 4 or 5).

- **Alignment** principle suggests that the elements of a design should be aligned with one or more other elements, creating a sense of cohesion and contributing to the design's aesthetics and perceived stability.

- **Symmetry** has long been associated with beauty. Starting from the symmetric shapes found in nature, human body, and later transmitted to architecture and design. Combinations of symmetries can create harmonious, interesting, and memorable designs.

- **Gutenberg diagram** is based on the observation that the reading gravity pulls the eyes from the

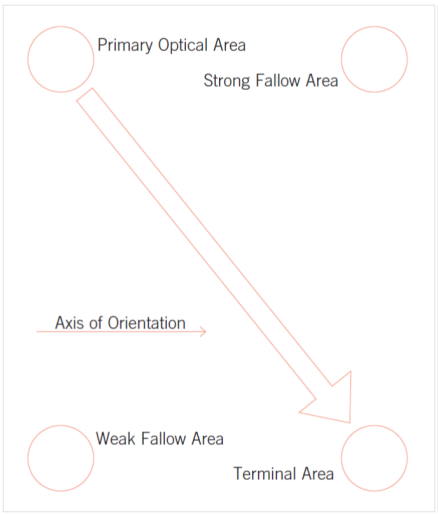


Figure 10.9: Gutenberg diagram [BHL03]

top-left to the bottom-right of the display (see Figure 10.9).[6] It recommends that the homogeneous elements in the design are organized in such manner to follow the reading gravity. In the case of heterogeneous elements the visual weight of each element will draw the attention and drive the eye movement, so the layout would not apply and could unnecessarily constraint the design.

- **Highlight** can be effective technique to bring attention to specific elements of the design. However, if not use with care, it can actually be counter productive and distract the reader, reducing the performance of the design. There are several guidelines on how the highlighting can be used effectively. For instance, only up to 10% of the visible design should be highlighted, **bold** should preferably used (as it distracts less the reader), but *italics* and underline could also be used for highlighting titles, labels, captions, and short word sequences. Color can also be an effective highlighting technique, but should be used less frequently and only in concert with other highlighting techniques. Other techniques like inverting and blinking can add more noise to the design and be even more distracting so they should be used only in the case when the objective is to highlight highly critical information.

- **Colors** can be used, as mentioned before, as an effective highlighting technique, but also to group elements or give them a specific meaning. Lastly, it can also improve the aesthetics of the design if used with care, but if used improperly it can seriously harm the form and the function of the design. There are two main concerns when it comes to using colors in our design: (a) *Number of colors* should be limited to what human eye can process at a glance (about five different colors). Moreover, it is strongly suggested that the design does not rely solely on the colors to transmit the intended information, given that a significant portion of population has limited color vision, (b) *Color combinations*, if used properly, can improve the aesthetics of the design. Either adjacent colors on the color wheel, the opposing ones, or at the corners of the symmetric polygon inside the color wheel (see Figure 10.10) are good choices. Such color combinations are typically found in nature.

- **Signal-to-Noise ratio** refers to the ratio of relevant to irrelevant information in the design, and intuitively it should be as high as possible. For instance, in Figure 10.11, the designs on the right are drastically improved, by removing the unnecessary design elements (i.e., those that do not convey any information) from those on the left.

- **Closure** refers to the tendency to perceive a set of individual elements as a single recognizable pattern, rather than considering them separately. Although the individual elements do not form

[6] Need to be aware that this is only true in occidental cultures and indo-european languages that follow such direction in writing.

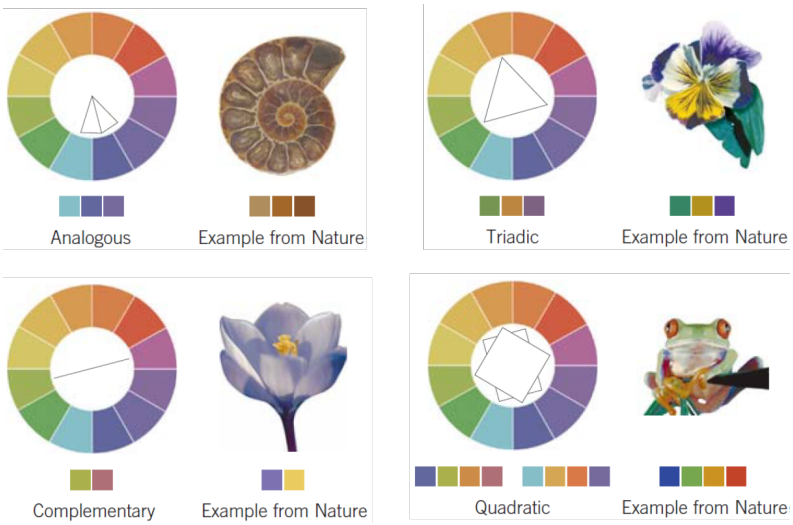


Figure 10.10: Color combination [BHL03]

a perfect shape of a pattern (see Figure 10.12), human brain has the ability to fill in the gaps and create a holistic picture, which results as a more interesting design. Involving closure can also reduce the complexity of the design, since in some cases we can only provide a sufficient number of elements to the design, while the brain creates the entire graphics.

- **Good continuation** is another principle that relates to the human perception of visual elements (i.e., Gestalt principles). Specifically, it states that if the visual elements of a design are aligned in some manner to form a proper geometry like line, human brain perceives them as a group and related. Using it in a design, good continuation makes easier to read the information. For instance, in Figure 10.13, the bar chart on the right is easier to read than that on the left as the end points of its bars form a straight line that is more continuous.

- **Five hat racks** refers to a principle that is used to order the elements in a design. More specifically, it states that there exist five ways of ordering information: category, time, location, alphabet, and continuum. The first three are self-explanatory, while continuum refers to organization of elements by magnitude (e.g., lowest to highest building in the world). The ordering in the design should be chosen wisely as different organizations dramatically influence which aspects of the information are emphasized.

- **Hierarchies** helps lowering the complexity of the design and transmitting an important information about the relationships between the elements in order to increase the knowledge about the structure of the system. The organization of hierarchies can be in the form of a (a) *Tree*, where child elements are found easily below or to the right of their parent elements, (b) *Nest*, where child elements are found within, contained by their parent elements, and (c) *Stair*, where several child elements are stacked below or to the right of their parent element.

- **Comparison** is a method used to illustrate a relationship or pattern inside or between systems. The information should be showed in a controlled way (i.e., using common measures and units, comparing benchmark variables of the system, and putting everything into the same context). For instance, the diagram in Figure 10.14 uses 12 wedges to show information (causes of death) for 12 months, and each wedge has three layers representing different causes of death (i.e., battle, diseases, and other). The graphs allows easy comparisons, representing the same variable (death rates) the same way (area of the wedge). Thus it is easy, just by a quick view, to notice that the main threat to the British army at the beginning was not the enemy, but diseases (e.g., cholera, dysentery, and typhus). We can also easily see the effects of the improved hygienic practices beginning in March 1855, when this ratio changed.

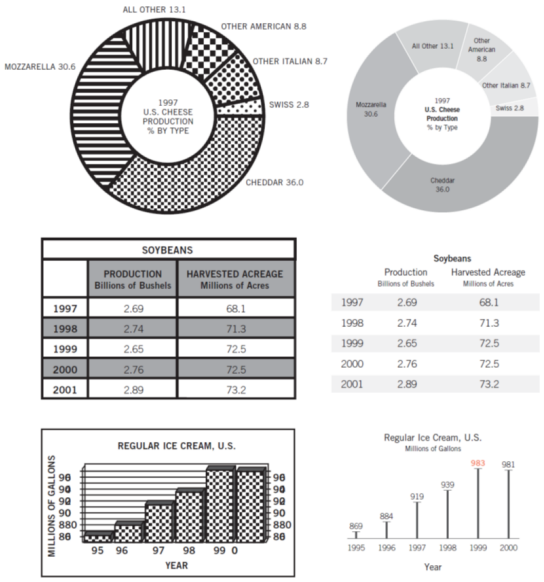


Figure 10.11: Signal-to-noise ration [BHL03]

Figure 10.12: Closure [BHL03]

- **Layering** is used to manage the complexity of the designs by organizing the information into related groupings and then showing only the elements of one group at a time. Layering can be either *two-dimensional*, in which case only the elements of one layer can be viewed at a time and they are revealed either in a linear (sequentially in a line) or non-linear way (as a tree or graph/web); or *three-dimensional*, in which case layers of information are placed on top of each other and multiple layes can be viewed at once (e.g., a geographical map with a layer of capital cities or a layer of temperature).

Lastly, there are several practical guidelines for having an effective dashboard design, stemming from the cognitive perception abilities of a person (see [Sta15] for details):

- Elements in the design should be logically and conceptually related so that they could be observed as a whole and easily compared.
- Number of elements should be at most seven (plus two if really necessary), which is the number of elements an average person can keep in the short term memory.
- Coloring in the design should be limited to a necessary minimum, extrapolating on the important information in the dashboard, but not serving as the only visualization method.

10.6 Multidimensional representation

When it comes to multidimensional data analysis, the possible graphical representations primarily depend on the number of dimensions that should be considered. For instance, one-dimensional analysis is concerned with analyzing only one data attribute or variable (visualizing its distribution) and appropriate forms are either histograms or boxplots (see Figure 10.15).

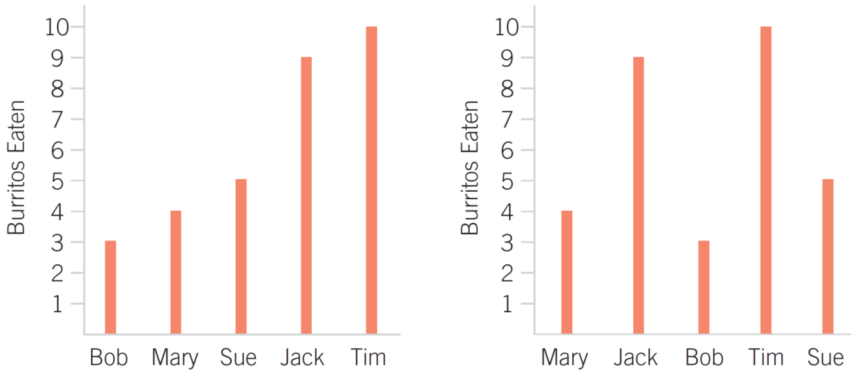


Figure 10.13: Good continuation [BHL03]

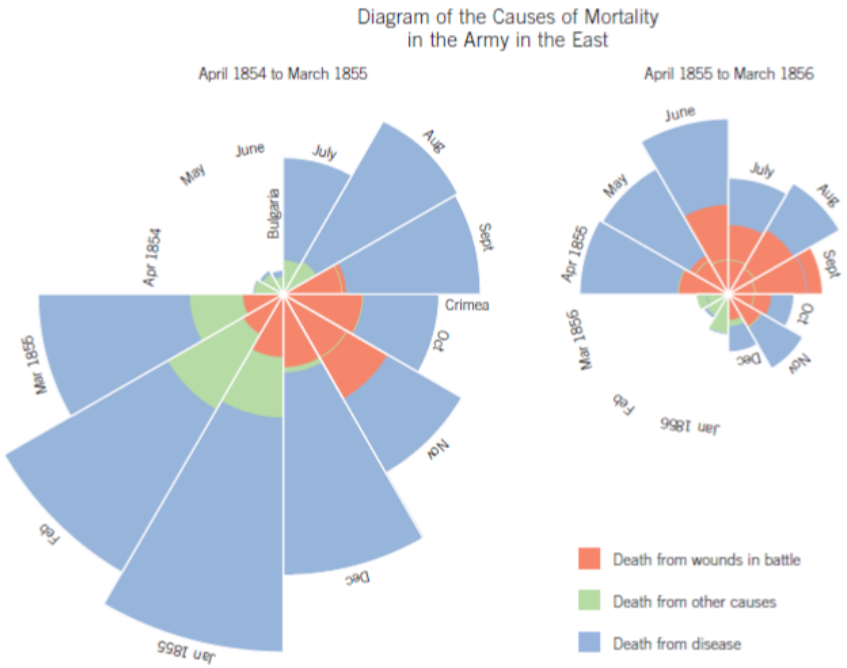


Figure 10.14: Coxcomb chart showing the causes of mortality of the British army in the east [BHL03]

With two-dimensional we can already employ more complex visualizations to analyze the potential pair-wise relationships or correlations amongst the different data attributes (e.g., 2D scatterplots or heatmaps). See an example in Figure 10.16. Such analysis can be extended to more dimensions (e.g., 3D scatterplot), either by introducing another (depth) dimension in the coordinate system (see Figure 10.17a) or by introducing other visual elements, like size (bubble chart), color, or hue. However, such visualizations can become more complex making them harder to comprehend by the viewer. Some of them like scatterplot, can be "unfolded" into a matrix so that all the data can be visualized (see Figure 10.17b).

Apart from considering the number of dimensions, the choice of an appropriate chart to visualize our multidimensional data can depend on other aspects of data themselves or the user analytical needs. To guide users in choosing the correct chart for their analysis, we can use a categorization method based on specific questions (see [Per15]). The following categories/questions could be considered:

- **Cardinality of the domain**, which when low (≤ 50) may indicate the use of charts that show all the individual values (e.g., trend lines or network), while when high would require more cumulative or aggregated values (e.g., bar chart, heat map).

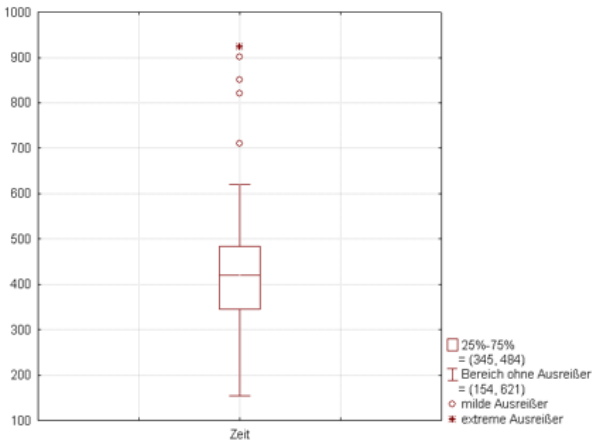


Figure 10.15: 1D: Boxplot (Wikipedia)

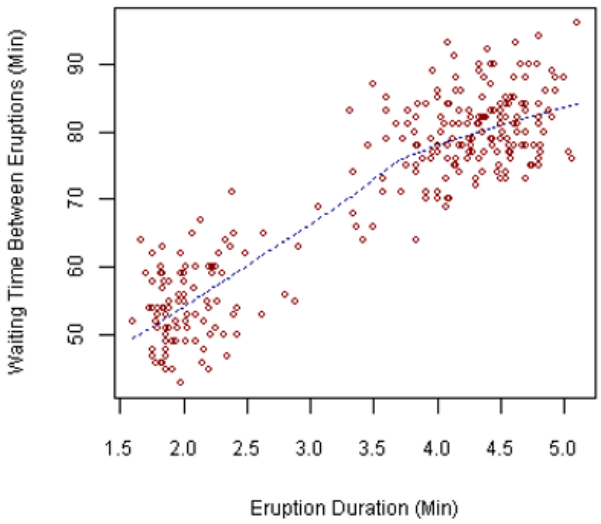
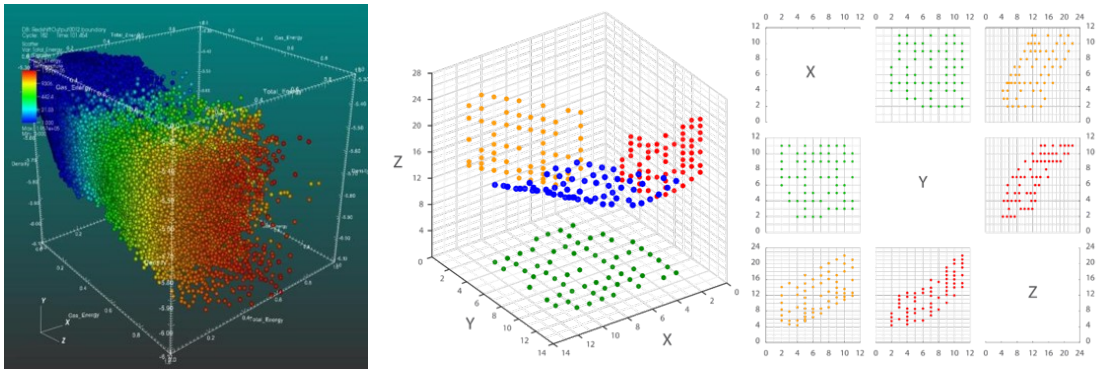


Figure 10.16: 2D: Scatterplot (Wikipedia)

- **Type of variable**, where *continuous* variables can be showed with charts like line charts, histograms while for *categorical* variables we can use for example bar chart, pie chart, or dendrogram[7].
- **Filtering** is useful when we are dealing with large amounts of data or when a user only wants to report on a specific subset of data that meets certain conditions. In such case, it is appropriate to use a chart that allows to filter data, by setting an interval or a threshold value.
- **Goal** of visualization, i.e., what information or message we want to convey to the viewer (e.g., *composition*, *distribution*, *comparison*, or *relationship* among different variables). Based on a specific goal, an appropriate chart should be chosen (e.g., using *dendogram* charts to visualize relationship among variables or *histograms* to show variable distribution).
- **Representation typology** should be chosen according to which *data type* the user wants to show, some charts can be more useful than others. For example, for showing location-based data it is better to use a geographical map, as well as the simplest way to display a hierarchy is through a hierarchical chart.

[7] https://en.wikipedia.org/wiki/Dendrogram



(a) 3D: Scatterplot (b) 3D: Scatterplot matrix

Figure 10.17: 3D scatterplot representations (Wikipedia)

Such categorization method for selecting the corresponding "best" choice chart for the specific data and user analytical needs is depicted in Figures 10.18 (for one dimension), 10.19 (for two dimensions), and 10.20 (for many dimensions).

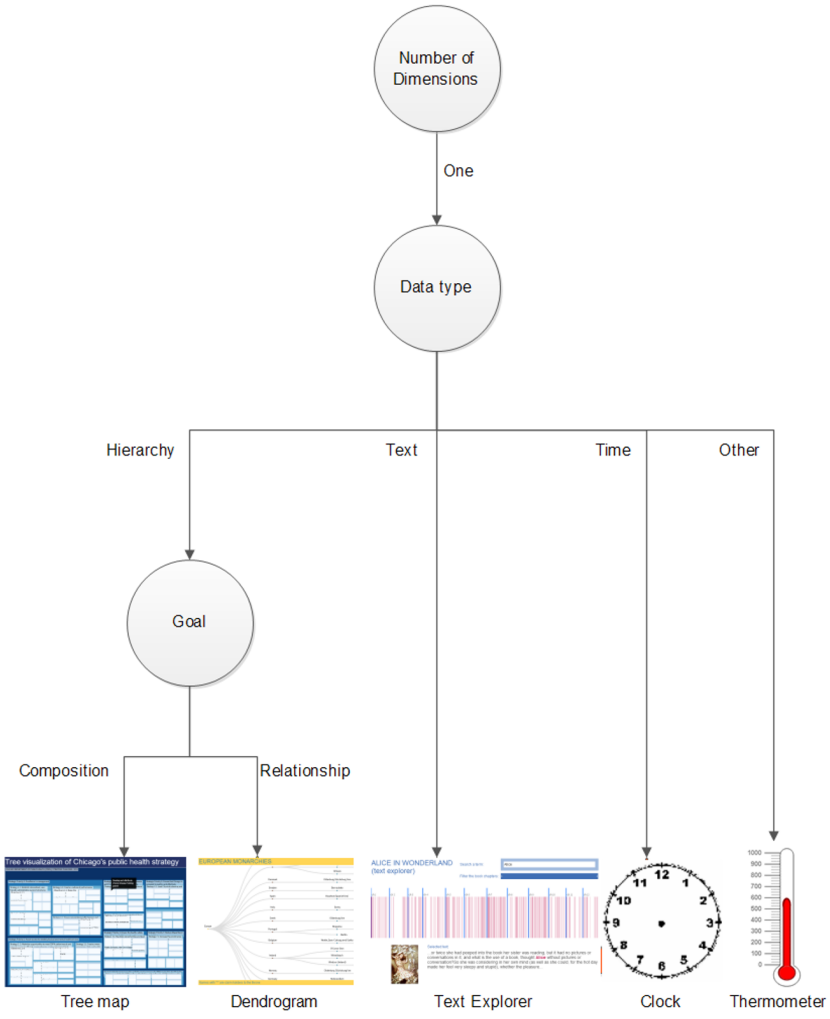


Figure 10.18: Selecting charts for one dimension data analysis [Per15]

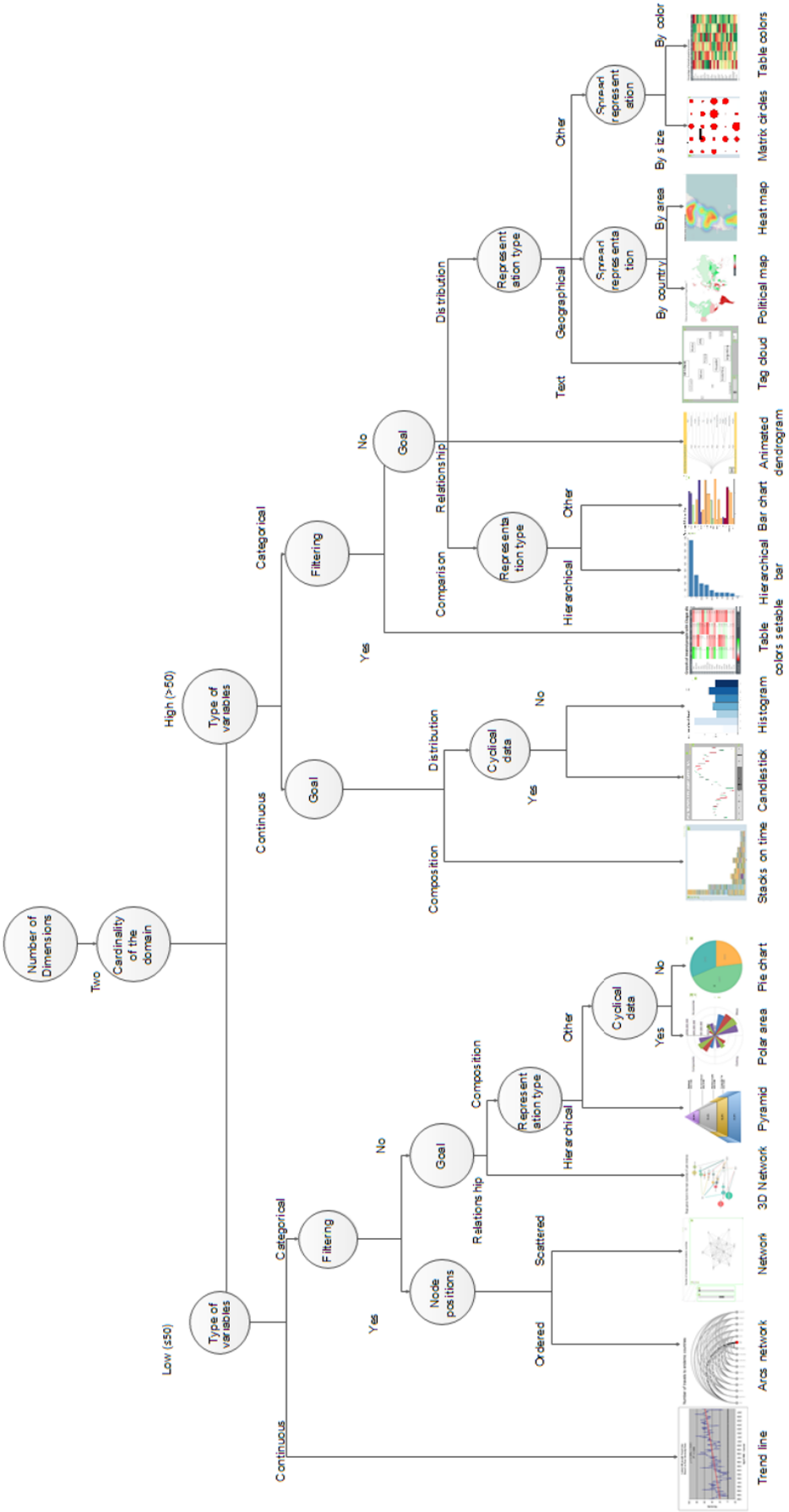


Figure 10.19: Selecting charts for two dimension data analysis [Per15]

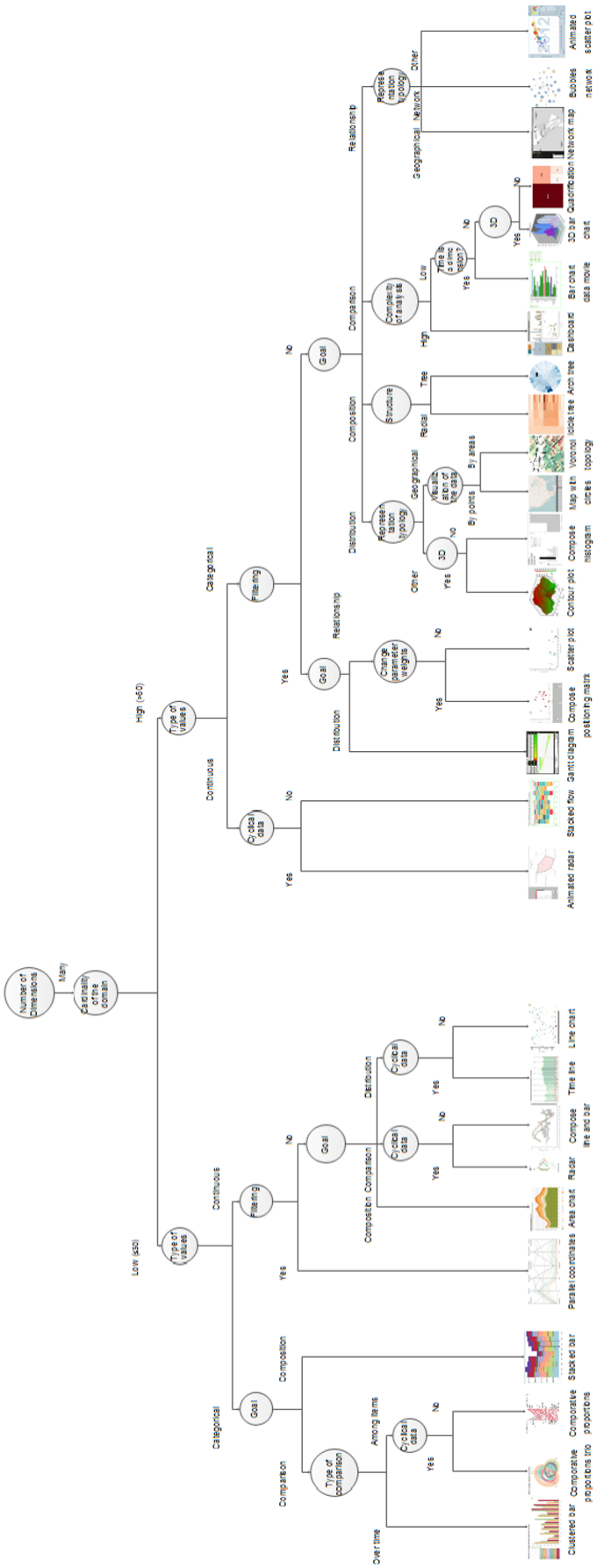


Figure 10.20: Selecting charts for many dimension data analysis [Per15]

Chapter 11

A (Brief) Introduction to Data Warehousing 2.0

With contributions of Oscar Romero

Data warehousing has been around for about three decades now and has become an essential part of the information technology infrastructure.[1] Data warehousing originally grew in response to the corporate need for information. Thus, a data warehouse is a construct that supplies integrated, granular, and historical data to the corporation. This simple definition has rendered extremely hard to match. Nowadays, after all the experiences gathered in data warehousing, authors start to distinguish between first-generation and next-generation data warehouses.

Basically, there are two main concerns behind this new paradigm: the inclusion of ALL the relevant data (from inside the organization and even from external sources such as the Web) and the REAL use of metadata. However, the second one is somehow a direct cause of the first one. After the success of data warehousing, the organizations widened their look towards data and wanted to include in their decision making processes alternative data such as unstructured data (of any kind, from plain text and e-mails to voice over IP), which makes even more difficult the data consolidation process. At this point is when metadata emerges as a keystone to keep and maintain additional semantics to data. Specifically, consider the following shaping factors:

- In first-generation data warehouses, there was an emphasis on getting the data warehouse built and on adding business value. In the days of first-generation data warehouses, deriving value meant taking predominantly numeric-based, transactional data and integrating those data. Today, deriving maximum value from corporate data means taking ALL corporate data and deriving value from it. This means including textual, unstructured data as well as numeric, transactional data.
- In first-generation data warehouses, there was not a great deal of concern given to the medium on which data was stored or the volume of data. But time has shown that the medium on which data are stored and the volume of data are, indeed, very large issues. In 2008, the first petabyte data warehouses have been announced by Yahoo and Facebook. Such amount of data is even beyond the limits of current Relational DBMS, and new data storage mechanisms have been proposed.[2]
- In first-generation data warehouses, it was recognized that integrating data was an issue. In today's world it is recognized that integrating old data is an even larger issue than what it was once thought to be.
- In first-generation data warehouses, cost was almost a non-issue. In today's world, the cost of data warehousing is a primary concern.

[1] This chapter is mainly based on the book "DW 2.0. The Architecture for the Next Generation of Data Warehousing", published by Morgan Kaufmann and authored by W.H. Inmon, Derek Strauss and Genia Neushloss, 2008 [ISN08].

[2] Namely, the NoSQL -Not Only SQL- wave has focused on the storage problem for huge data stores.

- In first-generation data warehousing, metadata was neglected. In today's world metadata and master data management are large burning issues. Concepts such as data provenance, ETL and data legacy are built on top of metadata.
- In the early days of first-generation data warehouses, data warehouses were thought of as a novelty. In today's world, data warehouses are thought to be the foundation on which the competitive use of information is based. Data warehouses have become essential.
- In the early days of data warehousing, the emphasis was on merely constructing the data warehouse. In today's world, it is recognized that the data warehouse needs to be malleable over time so that it can keep up with changing business requirements (which, indeed, change frequently).

All these issues represent challenges by themselves and still the community and major vendors are working on them. However, it is not part of this course objectives to gain insight, but the interested reader is addressed to [ISN08].

Bibliography

[BFI+22] Lukas Budach, Moritz Feuerpfeil, Nina Ihde, Andrea Nathansen, Nele Sina Noack, Hendrik Patzlaff, Hazar Harmouch, and Felix Naumann. The Effects of Data Quality on ML-Model Performance. *CoRR*, abs/2207.14529, 2022.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BHL03] Jill Butler, Kritina Holden, and William Lidwell. *Universal principles of design*. Rockport publishers Gloucester, MA, USA, 2010, 112–113., 2003.

[BPR+19] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. Data validation for machine learning. In Ameet Talwalkar, Virginia Smith, and Matei Zaharia, editors, *Proceedings of Machine Learning and Systems (MLSys)*. mlsys.org, 2019.

[BS16] Carlo Batini and Monica Scannapieco. *Data and Information Quality: Dimensions, Principles and Techniques*. Springer, 2016.

[CCS93] E. F Codd, S.B. Codd, and C.T. Salley. Providing OLAP (On Line Analytical Processing) to Users-Analysts: an IT Mandate. *E. F. Codd and Associates*, 1993.

[Che76] P. P. S. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[DGLLR07] Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. On reconciling data exchange, data integration, and peer data management. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 133–142, 2007.

[Do] Hong Hai Do. Data conflicts. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*, pages 565–569. Springer.

[DS15] Xin Luna Dong and Divesh Srivastava. Big data integration. *Synthesis Lectures on Data Management*, 7(1):1–198, 2015.

[FH76] Ivan P Fellegi and David Holt. A systematic approach to automatic edit and imputation. *Journal of the American Statistical association*, 71(353):17–35, 1976.

[FLM+99] Marc Friedman, Alon Y Levy, Todd D Millstein, et al. Navigational plans for data integration. *AAAI/IAAI*, 1999:67–73, 1999.

[FM18] Ariel Fuxman and Renée J. Miller. Schema mapping. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[GR09] M. Golfarelli and S. Rizzi. *Data Warehouse Design. Modern Principles and Methodologies.* McGraw-Hill, 2009.

[GSSC95] Manuel García-Solaco, Fèlix Saltor, and Malú Castellanos. *Semantic Heterogeneity in Multidatabase Systems*, pages 129–202. Prentice Hall International (UK) Ltd., GBR, 1995.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[Hal01] Alon Y Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[HHT05] Marios Hadjieleftheriou, Erik G. Hoel, and Vassilis J. Tsotras. Sail: A spatial index library for efficient application integration. *GeoInformatica*, 9(4):367–389, 2005.

[Inm92] W. H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., 1992.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[ISN08] W.H. Inmon, Derek Strauss, and Genia Neushloss. *DW 2.0. The Architecture for the Next Generation of Data Warehousing*. Morgan Kaufmann, 2008.

[Jar77] Donald Jardine. *The ANSI/SPARC DBMS Model*. North-Holland, 1977.

[Kim96] R. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley & Sons, Inc., 1996.

[KRTR98] R. Kimball, L. Reeves, W. Thornthwaite, and M. Ross. *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses*. John Wiley & Sons, Inc., 1998.

[Len02] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proc. of 21th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 233–246. ACM, 2002.

[LS09] A. Labrinidis and Y. Sismanis. View Maintenance. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3326–3328. Springer, 2009.

[May07] Arkady Maydanchik. *Data quality assessment*. Technics publications, 2007.

[McC87] Bruce Howard McCormick. Visualization in scientific computing. *Computer graphics*, 21(6), 1987.

[MTT18] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. Spatial indexing techniques. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[Pen05] Nigel Pendse. Market Segment Analysis. *OLAP Report*, Last updated on February 11, 2005. http://www.olapreport.com/Segments.htm (Last access: July 2009).

[Pen08] Nigel Pendse. What is OLAP? *OLAP Report*, Last updated on March 3, 2008. http://www.olapreport.com/fasmi.htm (Last access: July 2009).

[Per15] Daniele Perfetti. Business and visualization requirements for OLAP analysis of chagas disease data. Master's thesis, University of Bologna, Italy, 2015.

[PSZ06] Christine Parent, Stefano Spaccapietra, and Esteban Zimányi. *Conceptual modeling for traditional and spatio-temporal applications: The MADS approach*. Springer Science & Business Media, 2006.

[S+95] RT Snodgrass et al. The tsql2 temporal query. *Language*, 1995.

[Sat18] Kai-Uwe Sattler. Data quality dimensions. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*, pages 612–615. Springer, 2018.

[SBJS96] Richard T Snodgrass, Michael H Böhlen, Christian S Jensen, and Andreas Steiner. Adding valid time to sql/temporal. *ANSI X3H2-96-501r2, ISO/IEC JTC*, 1, 1996.

[Sta15] Miroslaw Staron. Dashboard development guide how to build sustainable and useful dashboards to support software development and maintenance. 2015.

[SV03] Alkis Simitsis and Panos Vassiliadis. A methodology for the conceptual modeling of ETL processes. In Johann Eder, Roland T. Mittermeir, and Barbara Pernici, editors, *The 15th Conference on Advanced Information Systems Engineering (CAiSE '03), Klagenfurt/Velden, Austria, 16-20 June, 2003, Workshops Proceedings, Information Systems for a Connected Society*, volume 75 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[SV18] Alkis Simitsis and Panos Vassiliadis. Extraction, Transformation, and Loading. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*, pages 1432–1440. Springer, 2018.

[SVS05] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. Optimizing ETL processes in data warehouses. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 564–575. IEEE Computer Society, 2005.

[SWCD09] Alkis Simitsis, Kevin Wilkinson, Malú Castellanos, and Umeshwar Dayal. Qox-driven ETL design: reducing the cost of ETL consulting engagements. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 953–960. ACM, 2009.

[The17] Vasileios Theodorous. *Automating User-Centered Design of Data-Intensive Processes*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona, January 2017.

[TL03] Juan Trujillo and Sergio Luján-Mora. A UML based approach for modeling ETL processes in data warehouses. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003, Proceedings*, volume 2813 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 2003.

[Vas09] V. Vassalos. Answering Queries Using Views. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 92–98. Springer, 2009.

[Vel09] Y. Velegrakis. Updates through Views. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3244–3247. Springer, 2009.

[VZ14] Alejandro A. Vaisman and Esteban Zimányi. *Data Warehouse Systems - Design and Implementation*. Data-Centric Systems and Applications. Springer, 2014.

Appendix A

Acronyms

BI Business Intelligence
CPU Central Process Unit
CSV Comma-separated Values
DAG Directed Acyclic Graph
DB Database
DBMS Database Management System
DDL Data Definition Language (subset of SQL)
DM Data Mart
DML Data Management Language (subset of SQL)
DW Data Warehouse
ELT Extraction, Load and Transformation
ER Entity-Relationship
ERP Enterprise Resource Planning
ETL Extraction, Transformation and Load
ETQ Extraction, Transformation and Query
FASMI Fast Analysis of Shared Multidimensional Information
FK Foreign Key
GUI Graphical User Interface
HOLAP Hybrid OLAP
I/O Input/Output
ISO International Organization for Standardization
JDBC Java Database Connectivity
LHS Left Hand Side
MBR Minimum Bounding Rectangle
MOLAP Multidimensional OLAP
NUMA Non-Uniform Memory Access
ODBC Object Database Connectivity
OLAP On-Line Analytical Processing
OLTP On-Line Transactional Processing
PDI Pentaho Data Integration
PK Primary Key
RDBMS Relational DBMS
RHS Right Hand Side
ROLAP Relational OLAP
SQL Structured Query Language
UML Unified Modelling Language

Appendix B

Glossary of terms

ACID: Transaction model adopted by all RDBMS and some NOSQL. It stands for:

- **A**tomicity: Either all operations inside the transaction are committed, or none of them is executed.
- **C**onsistency: If the database was consistent before the transaction, so it is after it.
- **I**solation: The execution of a transaction does not interfere with the execution of others, and vice-versa.
- **D**urability: If a transaction is committed, its changes in the database cannot be lost.

Access Path/Method: Concrete data structure (a.k.a. index) and algorithm used to retrieve the data required by one of the leaves of the process tree.

Access Plan: Procedural description of the execution of a query. This is the result of query optimization and is typically summarised/depicted in terms of a process tree.

ANSI/SPARC: The Standards Planning And Requirements Committee of the American National Standards Institute defined a three-level architecture in 1975 to abstract users from the physical storage, which is still in use in the current RDBMSs.

Architecture: A system (either hardware or software) architecture is the blueprint of the system that corresponds to its internal structure, typically in terms of independent modules or components and their interactions. It is said to be "functional" if it explicitly shows the functionalities of the system. An architecture can be centralized if the system is expected to run in a single location/machine, or distributed if it includes the components needed to orchestrate the execution in multiple and independent locations/machines. The architecture is said to be parallel if different components can work at the same time (i.e., not necessarily one after another). Distributed and parallel architectures are typically organized in terms of one coordinator component and several workers orchestrated by it.

Block: The transfer unit between the disk and the memory. This is also the smallest piece of disk assigned to a file.

Catalog: Part of the database corresponding to the metadata.

Data (or Database) Model: Set of data structures, constraints and query language used to describe the Database (e.g., Relational, Object-oriented, Semi-structured, Multidimensional).

Data Type: Kind of content stored in an attribute or column (e.g., integer, real, string, date, timestamp). Besides basic data types, we can also find complex (or user-defined) ones (e.g., an structured address which is composed by the string of the street name, the house number, zip code, etc.).

Database: Set of interrelated files or datasets (a.k.a. tables or Relations in the Relational model). According to its use, a database if called "operational" or "transactional" if it is used in the operation of the company (e.g., to register and contact customers, or manage the accounting), or "decisional" if it is used to make decisions (e.g., to create dashboards or predictive models). According to the location of the files, a database can be centralized if they reside in a single machine, or distributed if they reside in many different machines.

Database Management System (DBMS): Software system that manages data. The most popular one are those following the Relational model (a.k.a. RDBMS), or the more modern mixture or Relational and object features (a.k.a. Object-Relational DBMS). A Distributed DBMS (a.k.a. DDBMS) is that able to manage distributed databases. Otherwise, it is said to be centralized.

Entry: Smallest component of indexes, composed by a key and the information associated to it. In B-trees, we find entries in the leaves, while in hash indexes, the entries are in the buckets.

ETL: Data flow in charge of extracting the data from the sources, transforming them (i.e., formatting, normalizing, checking constraints and quality, cleaning, integrating, etc.), and finally loading the result in the DW.

Metadata: Data that describes the user data (e.g., their schema, their data types, their location, their integrity constraints, their creation date, their description, their owner). There are usually stored in the catalog of the DBMS, or in an external, dedicated repository.

Process tree: Representation at the physical level of the access plan of a query.

Relation: Set of tuples representing objects or facts in the real world. They are usually represented as tables and stored into files, but they can also be derived in the form of views. Each tuple then corresponds to a row in the table and a record in the file. Tuples, in turn, are composed by attributes, which correspond to columns in the table and fields in the records of the file. In statistical lingua, a relation represents a dataset containing instances described by features.

Relational Model: Theoretical model of traditional databases consisting of tables with rows (a.k.a. tuples) and columns (a.k.a. attributes), defined by Edgar Codd in 1970. Its main structure is the Relation.

Schema: Description of the data in a database following the corresponding data model (e.g., in the Relational model, we describe the data by giving the relation name, attribute names, data types, primary keys, foreign keys, etc.). We can find schemas representing the same data at the conceptual (i.e., close to human concepts and far from performance issues; e.g., UML), logical (i.e., theoretico-mathematical abstraction; e.g., Relational model) or physical (i.e., concrete data structures used in a DBMS; e.g., indexes and partitions in PostgreSQL) levels.

Selectivity [**Factor**]**):** Percentage of data returned by a query or algebraic operator, with regard to the maximum possible number of elements returned. We say that selectivity is "low" when the query or operator returns most of the elements in the input (i.e., the percentage is high).

Syntactic tree: Abstraction of the process tree in terms or Relational algebra operators.

Transaction: Set of operations over a database that are executed as a whole (even if they affect different files/-datasets/tables). Transactions are a key concept in operational databases.

Transactional: Refers to databases or systems (a.k.a. OLTP) based on transactions, and mainly used in the operation of the business (i.e., not decisional/analytical).