Bases de Datos

Sivana Hamer

Universidad de Costa Rica

Escuela de Ciencias de la Computación e Informática

CI-0127 Bases de Datos

Sivana Hamer
Escuela de Ciencias de la Computación e Informática, Universidad de Costa Rica

II 2022, v1.4.2

Importante: Este documento recopila contenidos de diversos de sitios web especializados, académicos y documentos compartidos por universidades. Toda la información es utilizada con fines estrictamente académicos. Para más información, puede ver las referencias.

Índice general

Parte I

Introducción

Capítulo 1

Introducción a las Bases de Datos

1.1. Data

Data are known or assumed facts. Nowadays, we generate and use data during our day to day. For example, while I am messaging someone with my phone lots of data is generated and consumed. Somewhere in my phone, there is a phone address that is accessed. Furthermore, if I'm using a messaging application there are servers with IP addresses that must be retrieved to send the message. The message itself also contains data, the message, who sent it, from where and when. Depending on the application other data related to the message may also be possible to recollect such as reactions (thumbs up), replies or who has seen it.

Having access to data has real world value. Companies build their entire business models by the information they store such as social networks. Furthermore, there are institutions that are in charge of creating and maintaining databases such as the national register of citizens or banks. These companies have an interleaved role within our society, having rights to decide who can vote, what they can buy or what products will they be aware of through advertisement. Some companies even lose their reputation by not managing data correctly. Thus, data is quite the commodity.

Throughout our history, we have always generated, used and preserved data. Due to the usefulness of computers we have migrated data stored in other mediums to a digital format. Specifically, data is stored in *databases* (DB) that are collections of data.

1.2. File databases

The first computer databases are assumed to be files. Databases can be stored in files such as a CSV. For example, if we define a database for students enrollment to courses. One file can store the students enrolled in the University with their ID, name and date of birth. In another file we can have all the courses that they have taken by storing a student, course and semester. We could even have another file with all the information about the courses. Though these databases are still common today, there are several disadvantages:

- **Data integrity:** Changes might cause errors in the integrity of the data. How can we ensure that the student name is the same for the enrollment data? How can we ensure that the semester is valid? What happens if a course changes their name?
- **Data access:** As the data has a certain value for the institution, they might want to retrieve certain information. How can we find all the years of birth of students

who have taken a course?

- **Data concurrency:** If there are several people trying to write to the file, we have to deal with concurrent changes. Do we allow concurrent changes? How do we choose which value to store? How do we reduce data anomalies?
- **Data durability:** Our file has to be stored in a computer somewhere. How do we manage the data if there is a crash during changes? How do we protect the computer from breaking?
- **Data security:** As our files are stored in a computer, someone without access might try to gather data. How can we ensure that the data is protected?
- **Data descriptiveness:** Let us assume that someone new is managing the file database, without training. How descriptive is the data? How descriptive are the data conventions? How descriptive are the data relationships?

1.3. Database Management Systems

Due to the previous disadvantages, Database Management Systems are widely used. A *Database Management System* (DBMS) is the software that manages a database. The goal of DBMS is to conveniently and efficiently store and retrieve data. These systems help:

- **Defining:** Stores the *meta-data* of the data in a *database catalog* or *dictionary*.
- **Constructing:** In charge of the process of storing the data.
- **Manipulating:** Allows functions to Create, Read, Update and Delete (CRUD) data.
- **Sharing:** Allows concurrent use of the database preserving integrity defined by certain rules.
- **Protecting:** Protects both from security threats and software malfunctions.
- **Maintaining:** Allows the database to evolve.

1.4. Database Systems

A *database system* is a database (collection of data) + DBMS (software that manages the data). Fig. 1.1 shows an example of a database system.

- A *query* is any interaction with the database. For example, retrieving all the students or adding a new enrollment.
- Someone interacts with the database requesting a query. For example, gathering all the students currently enrolled in a course or adding a new student. People can include *database administrators* (DBA) who are in charge of administering the database system, *programmers* that develop apps that access the database system, and *final users* who use an application that accesses the database system.
- This query is processed by the *query processor* that efficiently gathers the data.

- The data items related to the query are accessed and gathered from physical storage with a *storage manager*.
- The *meta-data* is data about the data (data types, structures and constraints). For example, allowing that year to contain only integers. As the meta-data is a collection of data about data (quite recursive) it is also a database (as seen in the Figure).
- The data represents the *stored data*. For example, a student entry with ID “123456789”, name “Ana Arias” and year of birth “1991”.
- A database *schema* describes the database design as in the the meta-data of the database.
- The *database state, snapshot or instance* is the actual current data in the database. For example, it would be all the students in the database at a moment in time.

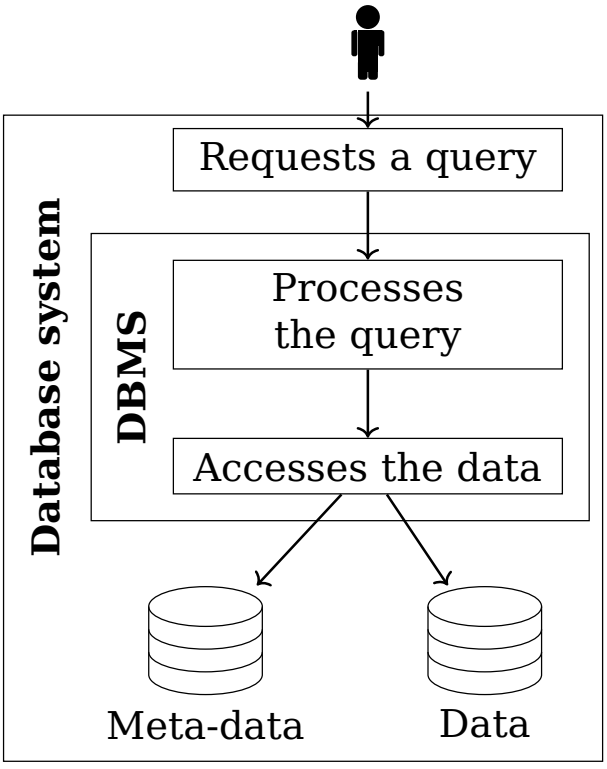


Figura 1.1: Example of a database system

1.5. Data abstraction

Database systems started out by combining both the physical storage of the data and the models to describe the data. However, this leads to maintenance issues leading to data model changes requiring updates for the DBMS physical storage. Furthermore, users needed to understand the physical storage of the data to interact with the database system.

Therefore, *data abstraction* has been defined to hide the finer grained details of the implementation based on the needs of the user, with database systems separating the abstraction into three different *abstraction levels*. There is an *independence* between

levels as their relationship is *mapped* between each other. The levels from lowest the lowest degree of abstraction to highest are:

- **Physical level:** Describes *how* the data is stored. Specifically, in this level the block of bytes that are stored are abstracted from the user.
- **Conceptual or logical level:** Defines *what* data is stored conceptually and *what* is their relationship through *data models* (Section 1.6). Though there is a relationship between the data and how it is physically stored, there is a *physical level of independence* as the details for the mapping are abstracted from the user.
- **External or view level:** Provides an interaction with the database for different users. There are different *views* defined for each user, limiting access to certain parts of the database. There is a *logical level of independence* as we can change what data we are accessing without having to change the logical level of the DBMS.

Though the data can be abstracted into higher or lower levels, the data only exists in the lowest abstraction level (physical level). A summary of all the levels with their relationships is shown in Fig. 1.2.

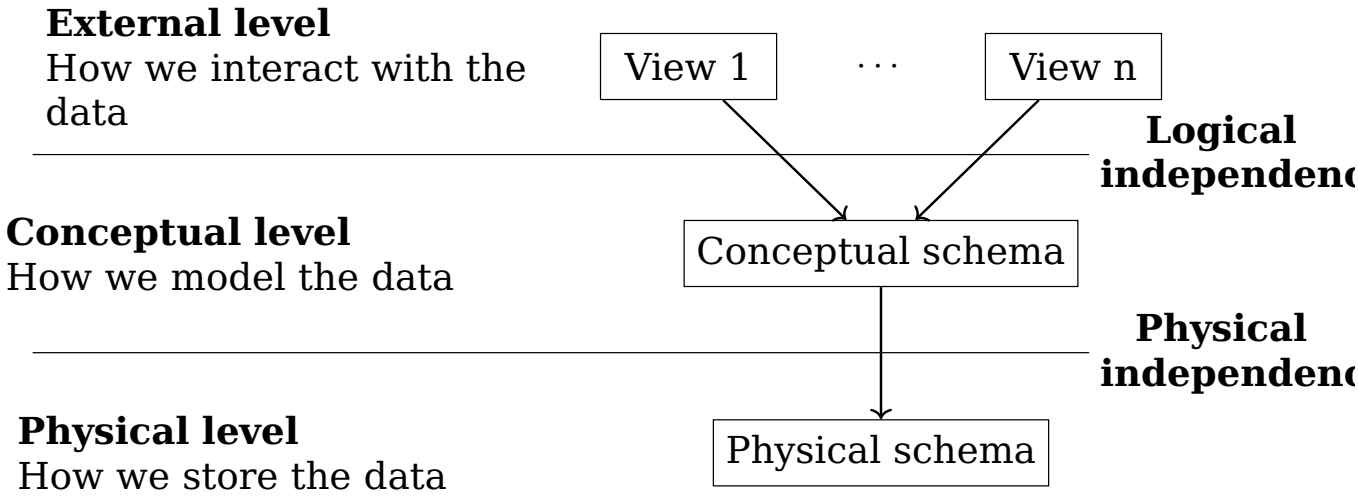


Figura 1.2: Levels of abstraction

1.6. Data models

A *data model* is a collection of concepts describing the database structure (data types, relationships and constraints) with a based set of operations to interact with the database. Data models can be categorized as:

- **High-level or conceptual:** Details how the users perceive the data. For example, the entity-relationship model.
- **Representationtal or implementation:** Describes an intermidiate model between how users perceive the data and how it is stored. For example, the relational, network, hierarchical and object-data models.
- **Low-level or physical:** Describes how files are stored in the computer.

- **Self-describing:** Combines bothe description of the data (schema) with the data values (state). For example, NoSQL or key-value.

Depending on the selected data model and/or the DBMS used, the database must be *designed*.

1.7. Database languages

To interact with the database we require specific languages. Though these languages can be categorized into different types, in practice they both form part of the same language. An example of a database language is SQL.

Data definition language

Data definition language (DDL) allows us to define the database schema (i.e., metadata). Specifically, DDL allows us to define the *conceptual* schema. DDL commands for SQL include CREATE TABLE, CREATE INDEX, ALTER TABLE, ALTER INDEX, DROP TABLE and DROP INDEX. An example of the command to define a student TABLE is the following:

```
CREATE TABLE STUDENT (
    ID CHAR(6) NOT NULL,
    NAME VARCHAR(30) NOT NULL,
    BYEAR DATE
);
```

Other schema defining variations of DDL include *Storage Definition Language* (SDL) to specify the internal schema and *View Definition Language* (VDL) to define the views with their mappings. There are no SDL systems for most relational DBMS and most DBMS use DDL to define the views. In SQL, one can define a view with CREATE VIEW or DROP VIEW. An example for a view in SQL that selects all the names of the students born in 1991 is the following:

```
CREATE VIEW STUDENTNAME (NAM)
AS SELECT NAME
FROM STUDENT
WHERE BYEAR = 1991;
```

Data manipulation language

Data manipulation language (DML) allows us to interact by retrieving, inserting, deleting and modifying the database. DML commands for SQL include SELECT, UPDATE, INSERT and DELETE. An example in that selects all the names of the students born in 1991 sis the following:

```
SELECT NAME
FROM STUDENT
WHERE BYEAR = 1991;
```

There are two types of DML:

- **Low-level or procedural:** Defines the data needed with *how* to get the data. This type is related to relational algebra.
- **High-level, declarative or non-procedural:** Defines the data needed without describing how to get the data. For example, SQL is of this type. This type is related to relational calculus.

Data control language

Data control language (DCL) defines who and what can access the database. In SQL, some DCL commands are GRANT and REVOKE.

1.8. DBMS architectures

Database systems can have different architectures such as centralized, client-server and n-tiered. The current most relevant architectures are tiered, using the database as the backend. The architecture can have multiple tiers:

- **Two-tier:** The application is in the client machine while the database resides in a server machine. The interaction between tiers is achieved with an application programming interface (API) that allows clients to call the DBMS. The standard API for DBMS is called Open Database Connectivity (ODBC).
- **Three-tiers:** The client machine acts as the front-end with an intermediate application server that stores the business logic. The application server thus interacts with the database server to retrieve the queries. Examples of these systems include web browsers or mobile applications.
- **n-tier:** One can define a variable amount of n tiers. Usually, the intermediate application server can be divided into different responsibilities to better distribute load.

The visual structure of two and three tier architecture is shown in Fig. 1.3.

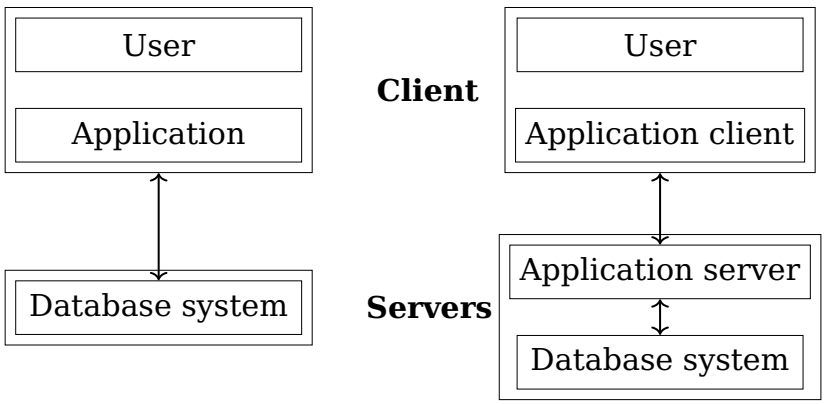


Figura 1.3: Two (left) and three (right) tier DBMS architecture

1.9. History

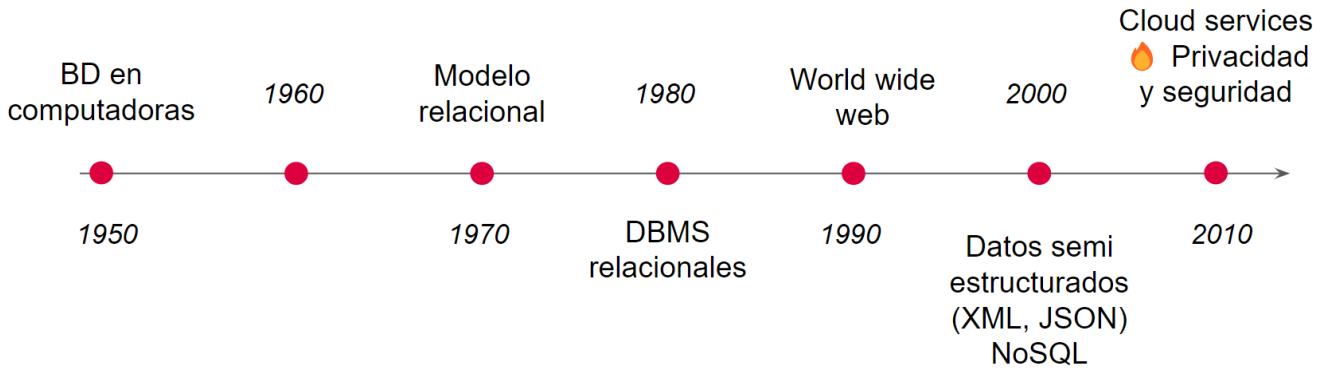


Figura 1.4: Summary of database history

1950s-early 1960s. Data was moved to computer data storage such as punch cards and tapes. There were several limitations with how the data could be read (sequentially) with some main memory being smaller than the data size.

late 1960s-early 1970s. Computers sequence limitations were lifted (no longer sequential). Relational data model began to be explored as they were defined in the 1970s by Edgar Codd.

late 1970s-1980s. DBMS started to use relational models as it became as efficient as other databases (hierarchical and network) with System R and easier to use than other databases. Initial research started for object oriented databases and distributed systems.

1990s. Database vendors added parallel capabilities and object oriented databases. The advent of the internet changed databases to be more query focused and process large amounts of data.

2000s. Advent of semi-structured data such as XML and JSON. Furthermore, several open-source database systems were developed. Due to the large amount of data, data mining became of interest and companies started to pilot NoSQL systems for data-intensive applications.

2010s. NoSQL became adopted for some systems (social networks) and had capabilities improved for users who interacted with the database. Data has been outsourced into cloud services. Concerns with users' privacy and security of the data have become hot topics.

Parte II

Diseño

Capítulo 2

Diseño conceptual

2.1. Database design process

The database design process starts by recollecting requirements and ends with the full database design. The complete process is shown in Fig. 2.1.

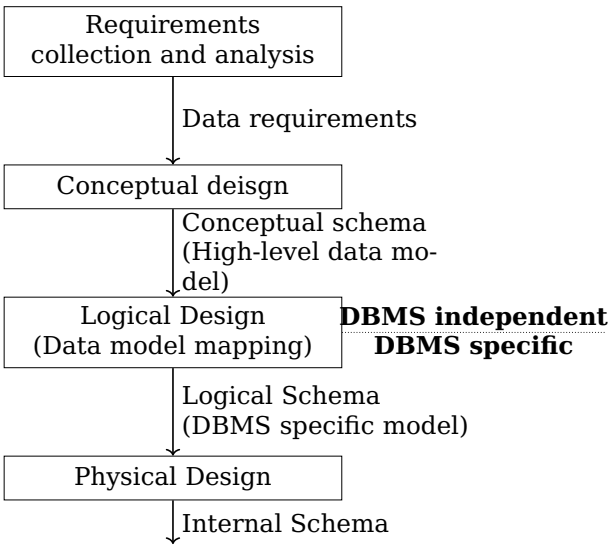


Figura 2.1: Database design process

1. To start the design, the database designers gather and document the *data requirements* from prospective database users. These requirements can be also gathered with the functional or non-functional requirements of the application, including the operations needed.
2. Based on the data requirements, the data high-level *conceptual design* is defined with a *conceptual schema* based on a data model. These designs do not include implementation details, being easier to understand for users and to create a good design due to not needing to be bogged down by the physical and implementation details. During or after this step, the operations in high level user queries can also be defined.
3. Using the conceptual schema, we can implement it by using a *logical design* or *data model mapping*. This implementation is DBMS specific and may be somewhat automated using tools.
4. Finally, the *physical design* can be defined. This includes the internal storage structure, file organization and indexes.

2.2. Entity relationship model

The *entity relationship* (E-R or ER) data model is a *high-level or conceptual data model*. In this model, data is described as entities (Section 2.3), attributes (Section 2.4) and relationships (Section 2.5). The model can be graphically represented in an *ER diagram*. For the following examples, we will try to exemplify using a course enrollment system for our university. The ER diagram shown in Fig. 2.2.

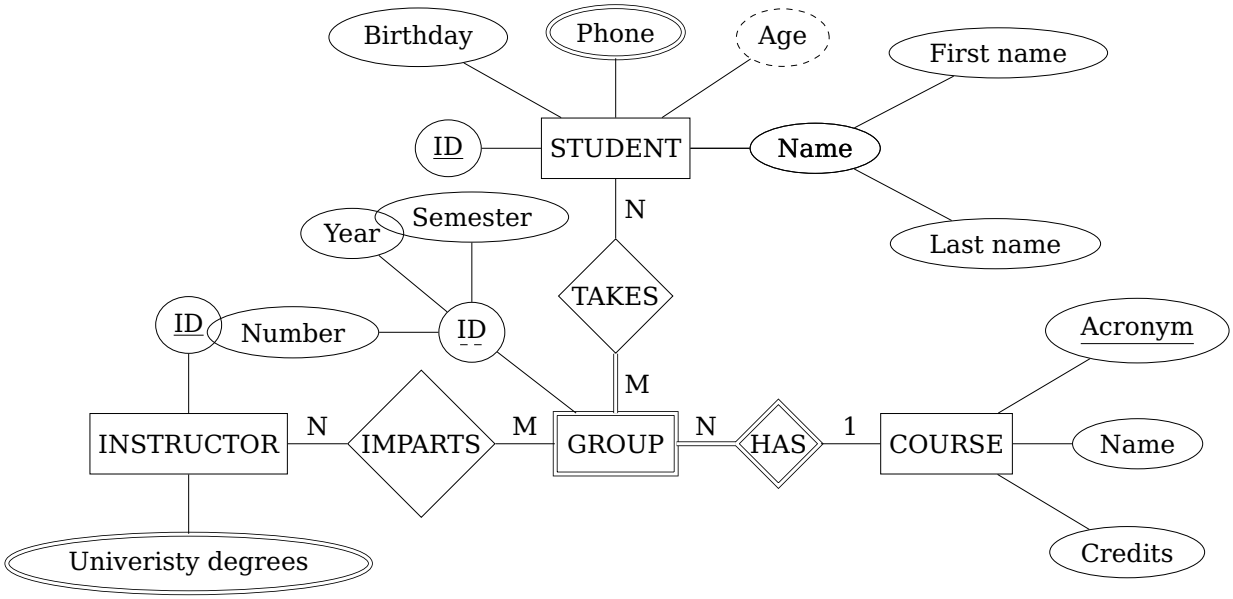


Figura 2.2: ER diagram for course enrollment

2.3. Entities

An *entity* is a real-world "thing" or "object" distinguishable from other entities. They may have a real world physical existence (e.g., person, building) or a conceptual existence (e.g., institution). In the course enrollment example entities can include instructors, courses, groups and students. An entity can be represented in an ER diagram as a square, as shown in Fig. 2.3. Inside the square a name is defined. Entities may be recognized as nouns.

Figura 2.3: ER diagram representation for an entity

While stored in a database, each entity will have different values for each attribute. For example, there might be instances for a person such as p_1 and p_2. *Entity type* are the set of entities that have the same attributes (e.g., PERSON, BUILDING). The *entity set* or *entity collection* are the collection of entities of a particular entity type in the database at a point of time (for PERSON the $p_1, \cdots, p_n$).

2.4. Attributes

Attributes are the properties that describe an entity. For example, for a person these could include the name, year of birth, or gender. Each attribute has a *value*.

For example, for the previous attributes a value for a person p_1 could be “Sivana Hamer”, 1900 and “female”, respectively. An attribute is represented by an oval as shown in Fig. 2.4. Attributes are connected by lines to the associated entity with their names inside the oval. Attributes tend to be the nouns describing another noun (entity).

Figura 2.4: ER diagram representation for an attribute

There are different types of attributes:

Simple vs Composite attributes. *Simple* or *atomic* attributes are not divisible in further parts (e.g, credits). While a *composite* attributes can be divided into smaller parts (e.g., name into first and last name). A composite attribute is represented by ovals in a hierarchical structure as shown in Fig. 2.5. Furthermore, a composite attribute can be further subdivided into more independ parts (e.g, first name into first and middle name). Composite attributes are used when a user refers to some of the smaller parts and sometimes the whole, but if only the whole is used there is no need to subdivide it into components.

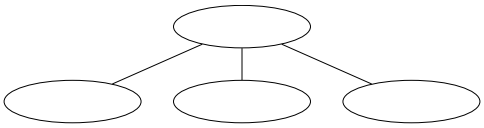


Figura 2.5: ER diagram representation for a composite attribute

Single-valued vs Multivalued attributes. For some attributes, each particular entity may have only a *single* value. For example, each person only has one birthday. However, there are other attributes that may have different numbers of values for an entity attribute (*multi valued*). For example, a student may be none, one or multiple phone numbers. There might be lower and upper bounds for the number of attributes. The ER diagram representation of a multivalued attribute is an oval with two lines, as shown in Fig. 2.6.

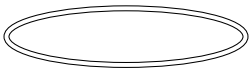


Figura 2.6: ER diagram representation for a multivalued attribute

Stored vs Derived attributes. A *stored* attribute is saved in the database, while a *derived* attribute is obtained from another attribute. For example, using a birthday attribute for student we can gather the age. The representation of a derived attribute is an oval with dashed lines, as shown in Fig. 2.7.

Figura 2.7: ER diagram representation for a derived attribute

Key vs non-key attributes. Every entity within a set has a unique *key* as a *uniqueness constraint*, called the *key attribute*. For example, every student has an ID that is unique to them. Another example is how every course has a unique

acronym in our university. The representation for a key value is shown in Fig. 2.8. A composite variable can be also used as a key, as long as each of the subparts together provide a unique value. For example, a license plate is unique as the combination of the state and the number. An entity may also have more than one key attribute, but always most have at least one. If an entity has no key attributes, it is a weak entity (Section 2.6).

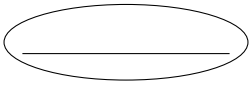


Figura 2.8: ER diagram representation for a key attribute

- Attributes might also save NULL if for an entity a particular value is not applicable (e.g., middle name if there is none) or if it is unknown (missing or is not known). For example, we might not know someones birthday.
- The values of the attribute can also be constrained between ranges and data types, though in the ER diagram this is not represented. This is called the *domain* of the attribute.
- A *complex* attribute is a multivalued and composite attribute.

2.5. Relationships

Relationships are associations between several entities. For example, the relationship between the entity STUDENT and GROUP is TAKES. Relationships might be recognized as verbs. It is represented in an ER diagram as a diamond, as seen in Fig. 2.9. Relationships are connected by lines to the associated entities, with the name inside the diamond.

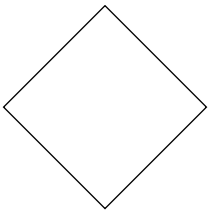


Figura 2.9: ER diagram representation for a relationship

- The *relationship set* represents all the relationships of the same type.
- The *degree of the relationship* is the number of associated entities to a relationship. For example, a binary relationship has two entities while a ternary has three associated entities. In our current ER diagram, we can only see binary relationships.
- Every entity that participates within a relationship has a particular *role*. These roles are described by role names, though they are not necessary for all the relationships.
- Relationships may be *recursive* or *self-referencing* when the same entity participates within a relationship as different roles. For example, a PERSON in a INSTRUCTing relationship may be the instructor or the apprentice. The representation of this relationship is shown in Fig. 2.10.

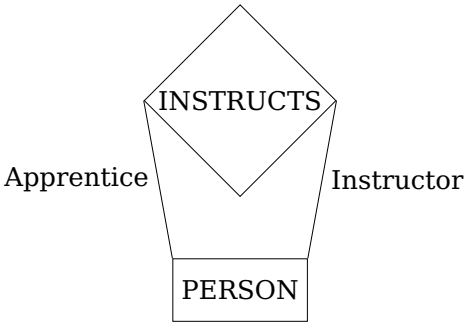


Figura 2.10: ER diagram for a recursive instructing relationship

- The *cardinality* determines the maximum number of entities and entities can be associated within a relationship set. The cardinality numbers are detailed above the connecting lines of the relationship and the entity. The cardinality can be used for any degree of relationship, though for this example we will use it for two entities in a relationship, A and B. They can be:

  One-to-one (1:1) Where every entity in A is associated at most with one entity in B (Fig. 2.11). For example, every STUDENT has one ID_CARD while every ID_CARD only has one STUDENT.

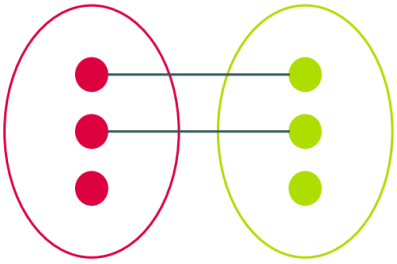


Figura 2.11: Cardinality 1:1 between entities A (red) and B (green)

  One-to-many (1:N). Where every entity in A can be associated with any number of entities (zero or more) in B, but B entities can be associated to at most one entity in A (Fig. 2.12). For example, a COURSE may be given in multiple GROUPs but every GROUP only has one COURSE.

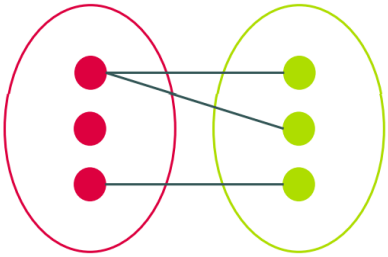


Figura 2.12: Cardinality 1:N between entities A (red) and B (green)

  Many-to-many (N:M). Where every entity in A can be associated with any number of entities in B and every entity in B can be associated with any number of entities in A (Fig. 2.13). For example, STUDENTs can take multiple GROUPs or even none if they just entered university. While, every GROUP is taken by multiple STUDENTs.

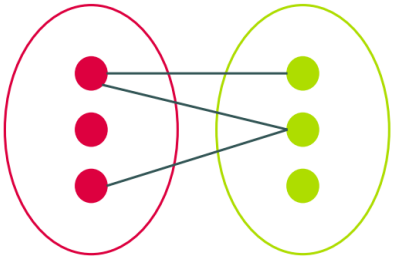


Figura 2.13: Cardinality N:M between entities A (red) and B (green)

- There may be *participation constraints* between entities in a relationship, representing the minimum number of relationships that may participate in a relationship.

  Total participation. Indicates that all entities must participate in the relationship. For example, every GROUP is expected to have a STUDENT (at least 7 in the UCR).

  Partial participation. Indicates that only some entities participate in a relationship. For example, every COURSE is not expected to have a GROUP as it might have not been given yet.

- Relationships may also have attributes. Though for 1:1 or 1:N the attribute may be associated to the entity with the 1 cardinality.

- We can represent the minimum and at most maximum relationships instances in the set using the (min, max) notation. We can see how we pass from min max notation to our ER diagram in Fig. 2.14.

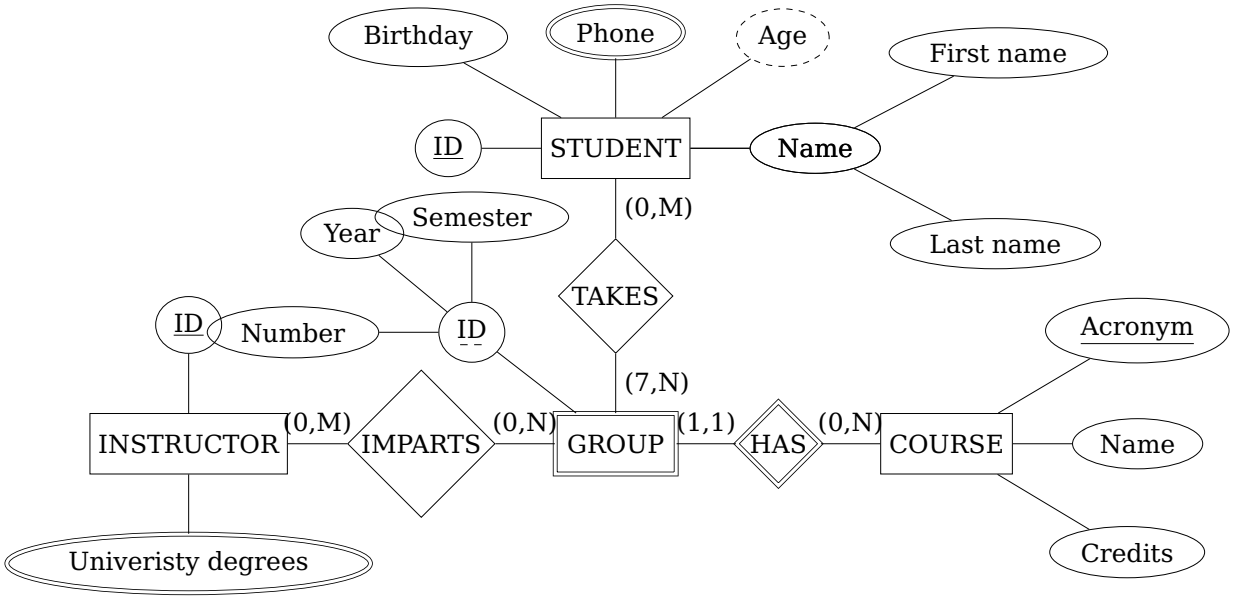


Figura 2.14: ER diagram for course enrollment using (min, max) notation

2.6. Weak entities

Entities without keys are *weak entities*, in contrast to entities with keys that are *strong entities*. Weak entities can be identified based on the combination of their attributes and another entity type (i.e., *identifying or owner entity type*). The relationship between the weak and owner entity is called the *identifying relationships*. An example of a weak entity is GROUP with the HAS relationship with COURSE, as every GROUP requires a course. The weak entity type and identifying relationship are represented in the ER diagram as a double lined box (Fig. 2.15) and diamond (Fig. 2.16), respectively.

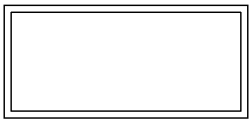


Figura 2.15: ER diagram representation for a weak entity

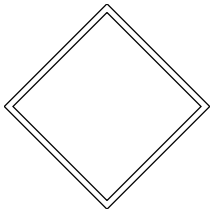


Figura 2.16: ER diagram representation for an identifying relationship

A weak entity has a participation constraint with their owner entity. Weak entities may require a *partial key* that in combination with the owner entity gives uniqueness. We do not require partial keys in 1 to 1 relationships. Partial keys are denoted with dashed underlining. We can see that for the GROUP, we have the group number, semester and year.

2.7. Binary versus higher degree relationships

While binary relationships are most common, sometimes there are situations where more than two entities must be involved in a relationship (*higher degree relationship*). As such, higher degree relationships are useful if all the entities are needed based on the semantics of the situations. Usually, three binary relationships are different from a ternary relationship. One can have both binary and higher degree relationships if they provide different meanings and they are all needed for the application. It is more efficient for the DBMS to have binary relationships.

For example, we can represent the relationship of a STUDENT TAKES a COURSE during a SEMESTER with a ternary relationship. The ER diagram is shown in Fig. 2.17. This represents a STUDENT s taking a COURSE c in a SEMESTER se, as a tuple of (s, c, se). A binary relationship between STUDENT and COURSE, (s, c), would only indicate that they TOOK a course in a semester (we can not be sure when). A binary relationship between COURSE and SEMESTER, (c, se), indicates that a COURSE was GIVEN but not who was it taken by. Finally, a binary relationship between STUDENT and SEMESTER, (s, se), indicates that they took classes in that semester but not which clases. We represented this relationship as binary in our above diagram by modeling the entities and relationships differently. This may be possible and preferable, depending on the semantics of the situation. We could have also added a

GROUP entity as a weak entity of COURSE and SEMESTER, however if we did that we would have no need for the ternary relationship as the information within the SEMESTER would be part of the GROUP. We must also must have the SEMESTER as part of the key in GROUP due to the fact that a group may have the same number, but be given in different semesters.

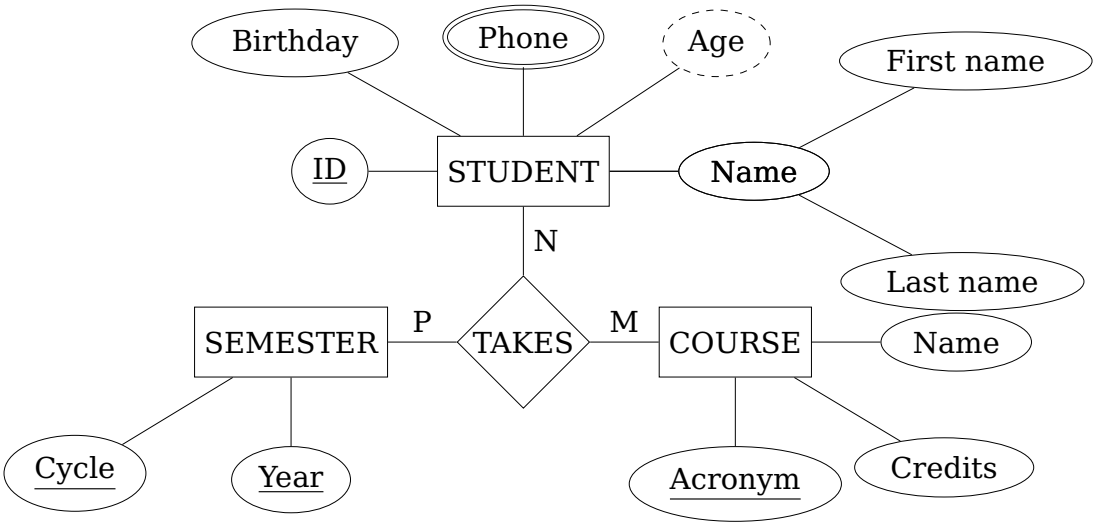


Figura 2.17: ER diagram for a ternary relationship

2.8. Naming conventions

- Names should be in singular, as they represent an item of a set.
- Entity and relationship type are written in all uppercase.
- Attributes names have only their initial letter in uppercase.
- Role names are all in lowercase.
- Diagrams are organized to be read from left to right and top to bottom.

2.9. Extended entity relationship model

The *extended entity relationship* (EER) data model extends the traditional ER model to better express the data in certain situations. The model adds the concepts of subtype (Section 2.10), specialization & generalization (Section 2.11) with their constraints (Section 4.4).

Four our example, we shall continue with the student example described previously. The EER diagram is shown in Fig. 2.18. Another EER notation is shown in Fig. 2.19.

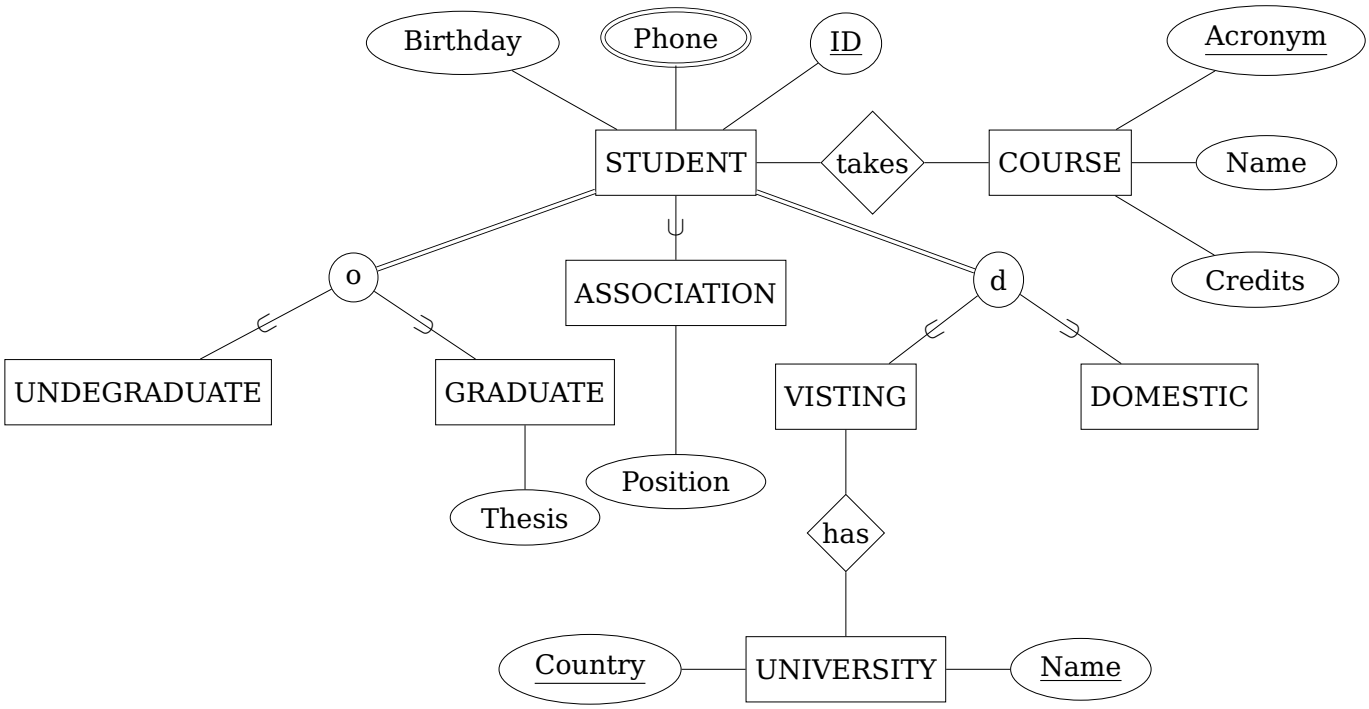


Figura 2.18: EER diagram for students

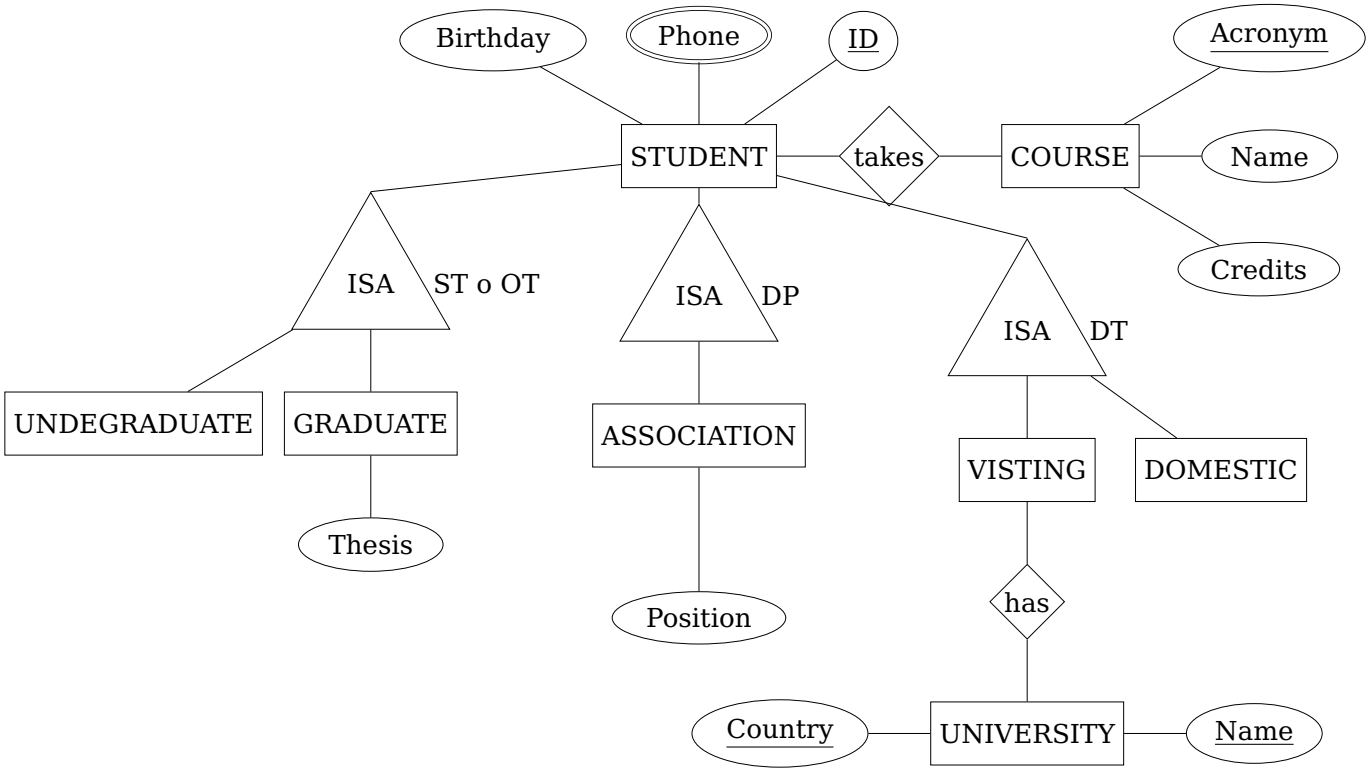


Figura 2.19: Alternative notation for EER diagram for students

2.10. Subtype

A *subtype* or *subclass* of an entity are subgroups for an entity type. For example, the STUDENT entity can be divided as UNDERGRADUATE (associates or bachelors), GRADUATE (masters or doctorate), DOMESTIC, VISITING or as part of the student ASSOCIATION. The different groups for an entity are the subtype, while the *superclass* or *supertype* is the higher level entity. This relationship is also called ISA (IS A or IS AN) type like a UNDERGRADUATE is a STUDENT. We can see that this is either represented by a line with a circle and another line defining the hierarchical structure between supertypes and subtypes, or with a triangle that has an ISA.

Each subtype may have different attributes, like GRADUATE students having a flag to indicate if they completed their dissertation or thesis. Subtypes may also have different relationships than the supertype, like VISITING student IS from another UNIVERSITY. These relationships may also have attributes. The attributes and relationships of the superclass are inherited by the subclass.

2.11. Specialization & Generalization

Specialization is the process where we define a *set of subclasses* for a *superclass*. For example, in our previous example we have three sets. First, we have the type of degree the student is getting, either being UNDERGRADUATE or GRADUATE. Second, the set where the students are getting the degree. They can be either DOMESTIC or VISITING from some other university. Finally, we have a set if the students are part of the ASSOCIATION of students. We can see that every set has a different circle or triangle with the ISA. We define these sets with these subclasses starting from the top (superclass) to the bottom (subclass).

Meanwhile, *generalization* is the opposite process where we define a superclass for a set of subclasses due to their similarity (in attributes and or relationships). For example, we could have defined UNDERGRADUATE and GRADUATE as entities and then have noticed that we could generalize certain attributes and relationships into one entity, STUDENT. We generalize the subclasses by analyzing the bottom (subclass) and determining their similarities at the top (superclass).

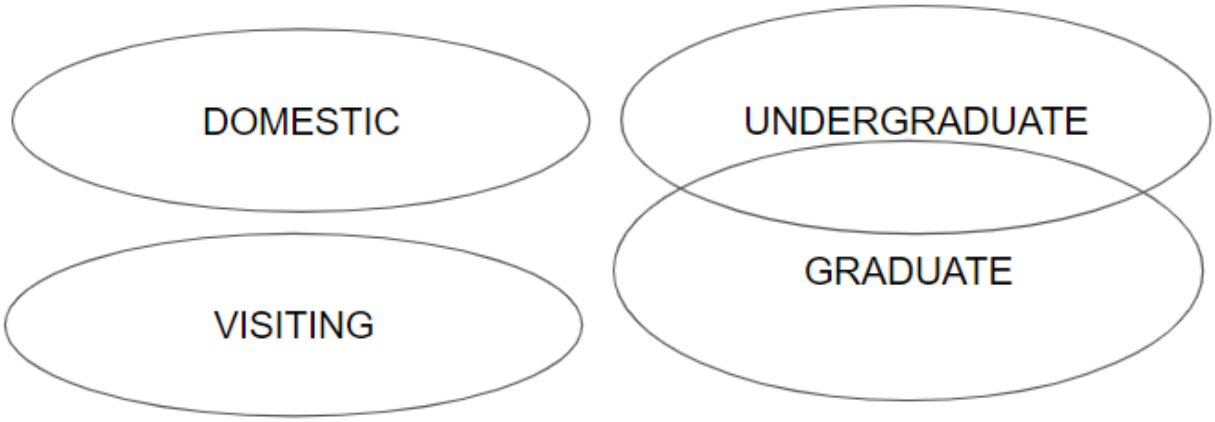
At the end of the day, defining which process was followed is somewhat subjective, so we shall not differentiate between both in our EER diagrams.

2.12. Constraints

There are different constraints we can define between the set of subclasses.

Disjointness or overlapping constraints: *Disjointness* defines that members of the subclass can only be one the subclasses of the specialization. For example, a STUDENT may either be DOMESTIC or VISITING but cannot be both (Subfig. 2.20a). We represent this using a d to denote disjointness. Meanwhile, *overlapping* entity types allow for an entity to belong to multiple types within the specialization. Such as someone can be both an UNDERGRADUATE and GRADUATE student at the same time (Subfig. 2.20b). We represent this using a o to denote overlapping. Sometimes, in Spanish we also use s for *solapamiento*.

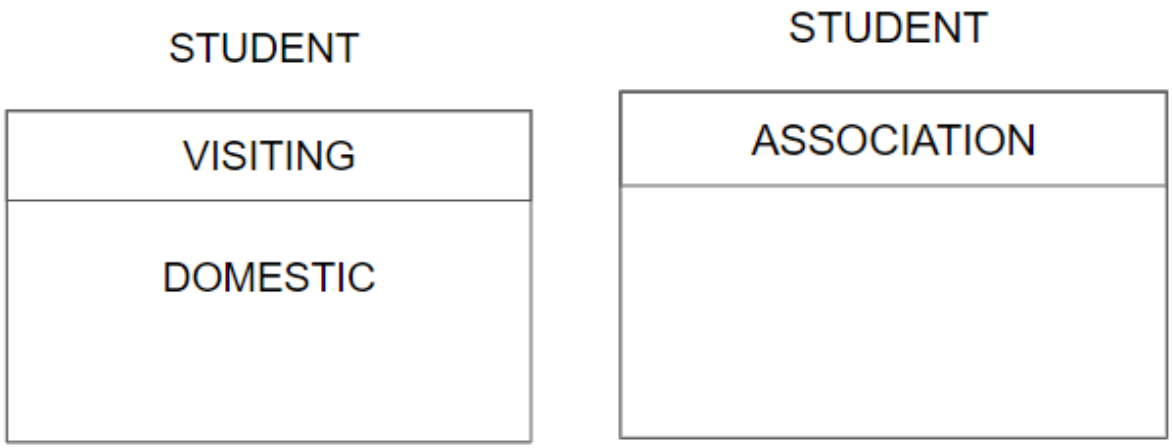
Totalness or partial constraints. *Totalness* determines if all entities in the superclass must be part of a subclass. For example, all STUDENTs must detail if they are either UNDERGRADUATE or GRADUATE (Subfig. 2.21a). This is denoted by



(a) Disjointness constraint (b) Overlapping constraint

Figura 2.20: Difference between disjointness or overlapping constraints

a double line between the circle and the superclass, or a t for total besides the ISA. Meanwhile, if it is not mandatory for the entities in the superclass to have a subclass, we have a *partial specialization*. For example, a student may be part of the student ASSOCIATION. This would be partial specialization as not all students would be part of the ASSOCIATION (only a select few as shown in Subfig. 2.21b). We denote partialness with only one line between the circle and the superclass, or with a p for partial besides the ISA.



(a) Total constraint (b) Partial constraint

Figura 2.21: Difference between total or partial constraints

As both types of constraints can be intertwined, the combinations shown in Table 2.1 are posible.

	Disjount	**Overlapping**
Total	DT	OT o ST
Partial	DP	OP o SP

Cuadro 2.1: Combinations of subclasses constraints

2.13. Review

Para cada uno de los siguientes requerimientos de datos, elabore el diseño conceptual utilizando modelo ER o EER. Las decisiones de diseño deben estar justificadas con base en los requerimientos. En caso de ambigüedad u omisión en los requerimientos, puede hacer suposiciones lógicas.

Simple

Sistema de bares

De: Dra. Alexandra Martínez
Parte 1:

- Se desea crear un sistema que almacene información sobre bares, cervezas y tomadores.
- Para cada bar se quiere registrar su nombre, dirección y teléfono.
- Para cada tomador se requiere almacenar su nombre, email y dirección.
- Para cada cerveza se requiere guardar su marca y nombre.
- Un bar vende muchas cervezas y una cerveza puede venderse en muchos bares.
- A un tomador le pueden gustar varias cervezas, y una cerveza puede ser tomada por muchos tomadores o ninguno.
- Un tomador frecuenta bares, y un bar puede ser frecuentado por varios tomadores.

Parte 2:
Modifique el diseño conceptual anterior (creado anteriormente) para incluir la siguiente restricción (requerimiento):

- Los tomadores sólo toman ciertas cervezas en ciertos bares.
  Ejemplo: el tomador Alex toma Bavaria en el bar Olafo's, Bavaria y Heineken en el bar Jazz Café, y Pilsen en Caccio's bar.

Analice si esta nueva restricción puede remplazar algún requerimiento de datos del ejercicio anterior.

Sistema de ligas

De: Dra. Alexandra Martínez

- Se desea crear una base de datos para almacenar información sobre ligas, equipos y jugadores.
- El nombre de cada liga es único.
- El nombre de los equipos no es único, pero ninguna liga tiene dos equipos con el mismo nombre.
- El número de los jugadores no es único, pero ningún equipo tiene dos jugadores con el mismo número.

Sistema de cartas al estudiante

Se quiere crear un sistema que permita guardar los libros asociados a una carta del estudiante.

- Cada docente imparte un curso dentro de un semestre.
- Cada curso impartido puede ser impartido en varios grupos.
- Cada curso impartido por la persona docente tiene una carta al estudiante para ese semestre en específico. Esto es debido a que las cartas al estudiante se pueden actualizar en cada semestre.
- Cada carta al estudiante tiene asociado varios libros.
- Cada libro tiene autores, una versión, un año de publicación, una editorial, y un título.

Sistema de adaptaciones

Se quiere crear un sistema para adaptaciones de libros a películas. Por ejemplo, el sistema debe permitir guardar información como los libros de Tolkien de “El señor de los anillos” fueron adaptados en la trilogía de películas dirigidas por Peter Jackson.

- Cada libro tiene un título, autor, año de publicación, un lenguaje de publicación y un género.
- Cada película tiene un nombre, una fecha de estreno, distribuidor y director.
- Cada libro puede formar parte de una serie.
- Una película puede ser adaptada de uno o varios libros. Pero, debe tener un libro para ser una adaptación.
- Un libro puede ser adaptado por varias películas. Por ejemplo, hay varías películas que son adaptaciones de “Alicia en el país de las Maravillas” como la versión animada de Disney y el live action de Tim Burton.

Sistema de genealogía

De: Dra. Alexandra Martínez

- Suponga que se quiere registrar una genealogía familiar en una base de datos.
- La información a almacenar sobre cada persona es: su nombre, su identiIicador (único), su madre, su padre y sus hijos (si tiene).
- Asuma que en el sistema solo se necesitan representar la madre y el padre biológicos de una persona. El sistema sería más complejo si consideramos, por ejemplo, la posibilidad de adopción.

Medianos

Sistema de bibliotecas BIBS

De: Dra. Alexandra Martínez

- La base de datos debe mantener información sobre las bibliotecas afiliadas, sus libros, ejemplares y usuarios.
- Cada biblioteca tiene un nombre único que la identifica, una dirección y al menos dos números de teléfono de contacto (el de recepción y el de préstamos).
- Para cada libro se debe almacenar su signatura, título, edición, año y autores. Una signatura identifica a un libro de forma única, pero no a sus ejemplares (pues todos tienen la misma signatura y no se podrían distinguir).
- Dado un libro específico, sus ejemplares se distinguen por medio del número de copia, que va desde 1 hasta el número total de ejemplares que hay en el consorcio de bibliotecas. Sin embargo, para distintos libros, pueden existir ejemplares con el mismo número de copia. Para cada ejemplar se debe poder consultar si está en préstamo o no.
- Una biblioteca puede tener uno o más ejemplares de un libro, sin embargo, un ejemplar debe estar asignado sólo a una biblioteca. Es posible que dos o más bibliotecas posean ejemplares del mismo libro, pero dichos ejemplares deben tener un número de copia distinto (es decir, no es posible que dos bibliotecas tengan la copia N del libro X).
- Cada usuario tiene un número de carné único (emitido por el consorcio de bibliotecas), nombre, email, teléfono y estado de morosidad. Un usuario está moroso si, para alguno de los libros que tiene prestados, ya pasó su fecha de devolución y aún no ha sido devuelto.
- Un usuario puede sacar (en préstamo) ejemplares de cualquier biblioteca del consorcio. Si un usuario saca varios libros a la vez, se registra un préstamo separado para cada uno de ellos, pues es posible que tengan fechas de devolución diferentes (algunos se prestan sólo por 2 semanas, otros por 3 meses, etc.). Para cada ejemplar que se da en préstamo se debe registrar la fecha de préstamo, la fecha esperada de devolución (periodo por el cual se le prestó el ejemplar) y la fecha real de devolución (fecha en que el usuario devolvió físicamente el libro). Si un usuario no ha devuelto un libro después de la fecha debida, se le pasa a estado moroso.

Sistema de supermercados SUPERK

De: Dra. Alexandra Martínez

- La base de datos debe mantener información sobre las diferentes sucursales de la cadena, los productos que ofrecen, el inventario y los clientes.
- La cadena SUPERK es multinacional, por lo que tiene sucursales en varios países, incluyendo Guatemala, Costa Rica y Panamá.
- Cada sucursal tiene un número que la identifica dentro de cada país, pero es posible que hayan dos sucursales con el mismo número en diferentes países. Para cada sucursal se debe almacenar su dirección y varios números de teléfono (por ejemplo, servicio al cliente, gerencia, información, etc.).

- Cada producto cuenta con un código único que lo identifica, además de marca, precio y descripción. Por ejemplo, la caja de leche Dos Pinos con 1 % de grasa se considera un producto distinto de la caja de leche Dos Pinos con 0 % de grasa, pero dos cajas de leche Dos Pinos con 1 % de grasa son el mismo producto para efectos del sistema (tienen el mismo código). Se debe llevar un inventario de productos por sucursal, que permita consultar y actualizar la cantidad en existencia de los productos por sucursal.
- Para cada cliente se requiere almacenar su nombre, email, teléfono y dirección. El email sirve como identificador del cliente. Algunos clientes poseen la tarjeta "super-ahorro", con la cual acumulan puntos y obtienen descuentos.
- Una sucursal pone a la venta muchos productos y un producto puede ser vendido en varias sucursales.
- Para cada compra que realiza un cliente, se debe guardar la fecha y hora en que se efectuó, la sucursal en la que se realizó, el monto total de la compra y el desglose de los productos comprados con su cantidad y precio unitario.

Sistema de Twitter

Se desea crear un sistema de redes sociales similar a Twitter.

- Cada usuario tiene un *handle*, nombre, descripción, imagen, contraseña, correo, verificación (check mark) y una fecha de creación de la cuenta.
- Cada Tweet tiene un usuario, una fecha agregada y texto.
- Cada Tweet puede ser *liked* por varios usuarios.
- Los usuarios pueden seguir a varios usuarios y pueden ser seguidos por varios usuarios.
- Cada usuario puede publicar varios Tweets.
- Cada Tweet puede ser *Retweeted* por otros usuarios.
- Cada Tweet puede ser respondido por otros usuarios. Este tweet de respuesta es un Tweet que puede contener información. Además, estas respuestas también se pueden responder.

Sistema de Spotify

Se desea crear un sistema de transmisión de *media* como Spotify.

- Cada usuario tiene un correo, nombre y una imagen de perfil.
- Dentro de la plataforma, los usuarios pueden reproducir distintos medios de comunicación (*media*). Para delimitar este sistema, se va a tener solo canciones y episodios de un podcast. Se guarda cuantas veces se reproduce el medio y el último segundo que se escuchó. Este último se requiere dado que un usuario podría querer continuar a reproducir el medio desde el último segundo en que escucho previamente.
- Cada canción tiene un nombre, artistas asociados, letra, duración y archivo.

- Cada álbum puede tener varias canciones. Un álbum tiene un nombre y una imagen. Además, el álbum está asociado a varios artistas pero siempre debe tener un artista.
- Cada artista tiene un nombre y una imagen de perfil.
- Cada playlist tiene varias canciones, un nombre, un usuario que lo creó y un estado de privacidad (públicos o privados).
- Cada podcast tiene un nombre, artista, descripción, imagen y varios episodios.
- Cada episodio tiene un nombre, un archivo, una fecha de publicación, una descripción, duración y un identificador.
- Cada usuario puede dar que le gusta un *media* o un álbum.
- Cada usuario puede seguir a varios artistas.

Sistema jurídico estadounidense

- La base de datos debe mantener información acerca del sistema jurídico estadounidense.
- Cada corte tiene un nombre, contado, estado, jurisdicción (contado, estatal o federal), ubicación y un número de teléfono. Cada corte puede ser reconocido por el conjunto de nombre, contado y estado.
- Cada estado tiene un nombre único, una abreviación única, una bandera única y un territorio en kilómetros cuadrados. Cada condado tiene un estado, un nombre y una población. Los condados se reconocen únicamente por el estado y su nombre.
- Cada juez@ tiene una cédula única (social security number), un nombre (primer nombre, segundo nombre y primer apellido) y una cantidad de años servidos. Una corte puede tener varios jueces, pero siempre debe tener mínimo uno. Se puede ser juez@ en varias cortes. El puesto de juez@ en una corte particular tiene una fecha de inicio y una fecha de fin.
- En las cortes, se litigan casos legales. Existen dos tipos de casos: civiles y criminales. En los casos civiles, un@ demandante demanda un@ acasud@ por una compensación monetaria entre partidos. En cambio, en los casos criminales un@ acusad@r demanda a un@ acasud@ para una penalización legal ante un delito. Por lo tanto, en los casos criminales se da una sentencia de meses. En los casos civiles la persona que demanda se llama un@ demandante mientras que en los casos civiles se llama un@ acusad@r, pero se refieren al mismo concepto.
- Cada caso se puede reconocer únicamente por la corte en que se litiga el caso, el apellido de la persona demandante o acusad@r, el apellido de la persona acasud@ y la fecha de inicio de la litigación del caso. Además, los casos también guardan la fecha final del caso y el veredicto.
- Dentro de cada caso, se presenta evidencia que son archivos asociados al caso. Cada evidencia se reconoce únicamente la combinación de un número y el caso en el que se asocia esa evidencia. Los casos pueden tener más de una evidencia.

- Cada persona que participa en el caso (demandante, acusad@r o acusad@) se llama un partido. Cada partido tiene una cédula única (social security number), un nombre (primer nombre, segundo nombre y primer apellido), una fecha de nacimiento, y una edad.
- Los partidos dentro de un caso tienen diferentes relaciones entre sí. Por ejemplo, puede ser una relación laboral, amistad, matrimonial, entre otros.
- Cada abogad@ tiene una cédula única (social security number), un nombre (primer nombre, segundo nombre y primer apellido), un tipo (transaccional o de litigación), una especialización (e.g., leyes de corporaciones, divorcio) y un bufete de abogad@s que pertenece. Además, cada abogad@ obtuvo su título máximo.
- L@s abogad@s representan a distintos partidos y un partido puede ser representado por muchos abogad@s, pero siempre debe tener un@ abogad@ como representación legal.

Sistema de criptomonedas

- La base de datos debe mantener información acerca de sistema que compra y venta de criptomonedas.
- Una criptomoneda representa a un tipo de corriente. En corrientes monetarias que utilizamos, es similar al dólar o colón. Cada criptomoneda tiene un nombre único, una abreviación única y una popularidad. Además, guarda el valor promedio de las ventas de las últimas 24 horas y si se encuentra actualmente con un precio que se está elevando o no.
- Un token representa a una unidad de una criptomoneda. Es similar a una moneda particular de colones. La única diferencia con una moneda normal es que no hay denominaciones distintas de corriente. Cada token se representa únicamente con un hash y la criptomoneda asociada. Una criptomoneda puede tener varios tokens, pero un token solo pertenece a una única criptomoneda.
- Existen distintos tipos de criptomonedas: *proof of work* y *proof of stake*. Las monedas basadas en proof of work requieren esfuerzo no significativo para la creación de la moneda. Generalmente, esto consiste en que una maquina tenga que utilizar fuerza bruta para desbloquear el siguiente token en la cadena. En cambio, las monedas basadas en proof of stake generan un porcentaje de tokens basado en la cantidad que se tiene de todos los tokens se tiene.
- Dentro de un blockchain, cada token tiene un token antecesor dentro de la cadena. Además, podría tener un sucesor si es que ya se creó el siguiente token.
- Cada cuenta guarda cuales tokens tiene comprados. Un token solo tiene una persona dueña a la vez y puede ser comprado varias veces.
- Cada cuenta tiene un país asociado, dado que se regula bajo las leyes de esa nación. Los países tienen un nombre único, un código de teléfono único (e.g., 305 en Costa Rica) y una bandera única.
- Dentro de nuestro sistema, existen diversas personas usuarias que pueden utilizar el servicio. Cada cuenta de una persona usuaria tiene un nombre único, un correo único, una contraseña y una cantidad de fondos en dólares que tiene disponibles. Las personas usuarias almacenan también una cantidad de tokens totales de las cuales son duenas.

- Cada cuenta puede adquirir distintos tipos de tokens de criptomonedas en una transacción. Cada transacción tiene tokens asociados, la cuenta que realizó la compra y la cuenta que realizó la venta. Una transacción puede tener varios tokens y un token puede ser comprado en varias transacciones. Además, para cada token de un tipo dentro de una transacción se tiene un precio de compra dado en dólares.

Capítulo 3

Diseño lógico relacional

3.1. Relational model

The relational model is a logical model based on relations. As it is a logical model, data is not stored physically as it is represented in the model. As an example, we shall use the SCHOOL relation shown in Fig. 3.1 that represents the SCHOOLs within our university with some fictional data and suppositions.

Name	Acronym	Phone_number	Number_students
Ciencias de la Computación e Informática	ECCI	2511-8000	888
Ciencias de la Comunicación Colectiva	ECCC	2511-3600	999
Lenguas Modernas	ELM	2511 8391	NULL
Administración de Negocios	EAN	2511-9180	3000
Antropología	EAT	2511-6458	500
Matemática	EMat	2511-6551	1500

Figura 3.1: SCHOOL relation

Relations

A *relation* or *"relationship*" (not really the correct term) is an unordered set of attributes. We can think of a relation as an unordered *table* with columns (i.e., attributes) and rows (i.e., tuples). The order of both the attributes and rows does not matter (with some exceptions).

A *relational scheme, schema or intention* defines a relation. It is represented as $R(A_1, A_2, \cdots, A_n)$

- R is the relational name. We write relational names in upper case. Common relational names include Q, R and S.
- A_i represents an attribute.
- The *degree or arity* of a relation is the number of attributes A_i.
- The *cardinality* is the total number of combinations of tuples (cartesian product $\times$) with values $|D|$ possible for all attributes in the relation.

Example: We can represent the relation schema of $SCHOOL = S$ as either of the following:

$$SCHOOL(Name, Acronym, Phone\_number, Number\_students) =$$

$$S(Name : string, Acronym : string, Phone\_number : string, Number\_students : integer)$$

The relation has a degree of four as there are four attributes.

A *relation state or extension* describes the tuples currently stored in the relation. It is represented as $r = r(R) = \{t_1, t_2, \cdots, t_m\}$.

- r is the current state of the relation. We write relational states in lower case. Common relational states include q, r and s.
- R is the relational name of the relational schema.
- t_j represents a tuple. Tuples have not specified orders within the relation.

Example: We have several tuples within the SCHOOL relation (non gray rows).

Attribute

An *attribute* A_i describes an aspect or role of an unordered set of values within a relation R. Each attribute has a *domain* of possible values within the column. The relation and column name provides meaning to the set of values within the relation.

Example: There are several attributes for each school within our school. These include:

- **Name:** The name of the school.
- **Acronym:** The acronym of the school.
- **Phone_number:** A phone number to contact the school. A school may have multiple phone numbers.
- **Number_students:** The number of students that are currently enrolled in the school.

Domain

A domain $D = dom(A_i)$ details a set of atomic (indivisible) values for attribute A_i. As values are atomic, composite and multivalued attributes are not allowed in the model, but can be integrated with additional representations.

For a domain we can describe the logical definitions, data type and format. We can also define a range of possible values. The number of values within the domain $dom(A_i)$ can be represented as $|dom(A_i)| = |D|$.

Example:

- **Name:** A string with no specific format.
- **Acronym:** A string that begins with a capital E. No school can have an acronym larger than four characters.
- **Phone_number:** In Costa Rica, phone numbers are strings of the form $nnnn - nnnn$, with n as an integer value. The first n cannot be 0.
- **Number_students:** A positive integer with a value above 0.

Tuple

A *tuple* represents a collection of related data values represented as $t = \{v_1, v_2, \cdots, v_m\}$.

- t represents the tuple. We write tuples in lower case. Common tuple names include t, u and v.
- v_i is a value within $dom(A_i)$ or a NULL value.
- NULL values can represent unknown values, values that do not apply or values that were not retrieved. However, it has been difficult to represent different types of NULL values in the relational model. Two NULL values do not mean that both tuples are the same. It is best to avoid NULL values if possible.
- Values do not need to be ordered as long as the relationship between attributes and values is maintained.
- $t[A_i] = t.A_i$ refers to the value v_i of attribute A_i for tuple t. We can represent multiple attributes as a list separated by commas.

Example: For the tuple of our computer science school:

$$t_{ECCI} =< \text{Ciencias de la Computación e Informática}, \text{ECCI}, 2511 - 8000, 888 >$$

3.2. Keys

Due to integrity constraints, relations have different types of keys for attributes.

Superkey

A *superkey* SK specifies the set of attributes that no two tuples in R can have the same values. Therefore, it specifies *uniqueness*. Every relation must have one superkey and they may have redundant attributes.

Example: We can define as a SK the attributes Name and Acronym. Another SK can be Phone_number as no school can use the same number.

Key

A *key* of R is a superkey that has the additional property that removing any additional attribute from the set of attributes of the superkey will make it no longer a superkey. Thus it satisfies properties for *uniqueness* and *minimalism*. All keys are superkeys, but not all superkeys are keys.

Example: The set of Name and Acronym is not a key. It is not minimal as we could remove either Name or Acronym. However, Phone_number is a key as it is unique and we cannot remove any attribute while maintaining its uniqueness property.

Primary key

A relational schema can have more than one key, with each key being called a *candidate key*. One of the candidate keys is designated as the *primary key* that is used to identify the relation. We represent this by underlining the attributes of the key in the relational schema. The non-primary candidate keys are considered *unique keys* that are not represented in the model.

Though in theory we can select any candidate key as the primary key, it can be recommended to select an integer as they are faster to find while stored. However, the relational model does not really recommend this as it is supposed to represent business rules.

Example: Both Acronym and Phone_number are candidate keys. We can select the Acronym as the primary key, leaving Phone_number as a unique key.

Foreign key

A foreign key in R_1 references a relation R_2 by storing the primary key of R_2 in an attribute in R_1. Thus, there is a relation between the tuples of R_1 and R_2 that have the same key.

Example: SCHOOL could have another attribute that references the DEPARTMENT of each school. Thus, SCHOOL could have a foreign key attribute that references DEPARTMENT.

3.3. Constraints

For our database, we shall define several *restrictions or constraints* on the values in the database states due to business rules or model restrictions.

Inherent model-based or implicit constraints

Constraints implicitly inherited by the data model. These constraints are defined by the constraints based on the model described in Section 3.1.

Schema-based or explicit constraints

Constraints explicitly defined in the schema of the data model via DDL (Data Definition Languages).

Domain constraints. For each tuple t, the value v_i of attribute A_i must be atomic and from the domain of A_i.

Key constraints. All tuples within a relationship must be unique, as defined by a Super Key SK. Thus for every pair of tuples t_1 and t_2 within a relation they have to be distinct, described as:

$$t_1[SK] \neq t_2[SK]$$

Not NULL constraints. Attributes define if NULL values are permitted and must be followed.

Entity integrity constraints. No primary key can be NULL, as it is used to identify individual tuples in the relation.

Referential integrity constraints. A foreign key in relation R_1 that references relation R_2 satisfies this constraint if the primary key attributes of both relations have the same domain and R_1 references an existing tuple of R_2.

A database must be in *valid* in every relational state for every relational schema based on the integrity constraints. A database that does not obey all integrity constraints is *not valid*.

Application-based or semantic constraints

Constraints that **cannot** be explicitly expressed in the data model schema. These must be informed either in the application or with SQL (triggers or assertion). As business rules may be complex to implement in the database, it is common that they are specified in the applications. However, constraints should always be defined at the database level if possible. One could also implement these constraints at several levels to ensure the validity of the database.

3.4. Mapping ER

For most of the following examples for mapping between the ER to the relational model, we shall use Fig. 3.2 (variation of Fig. 2.2).

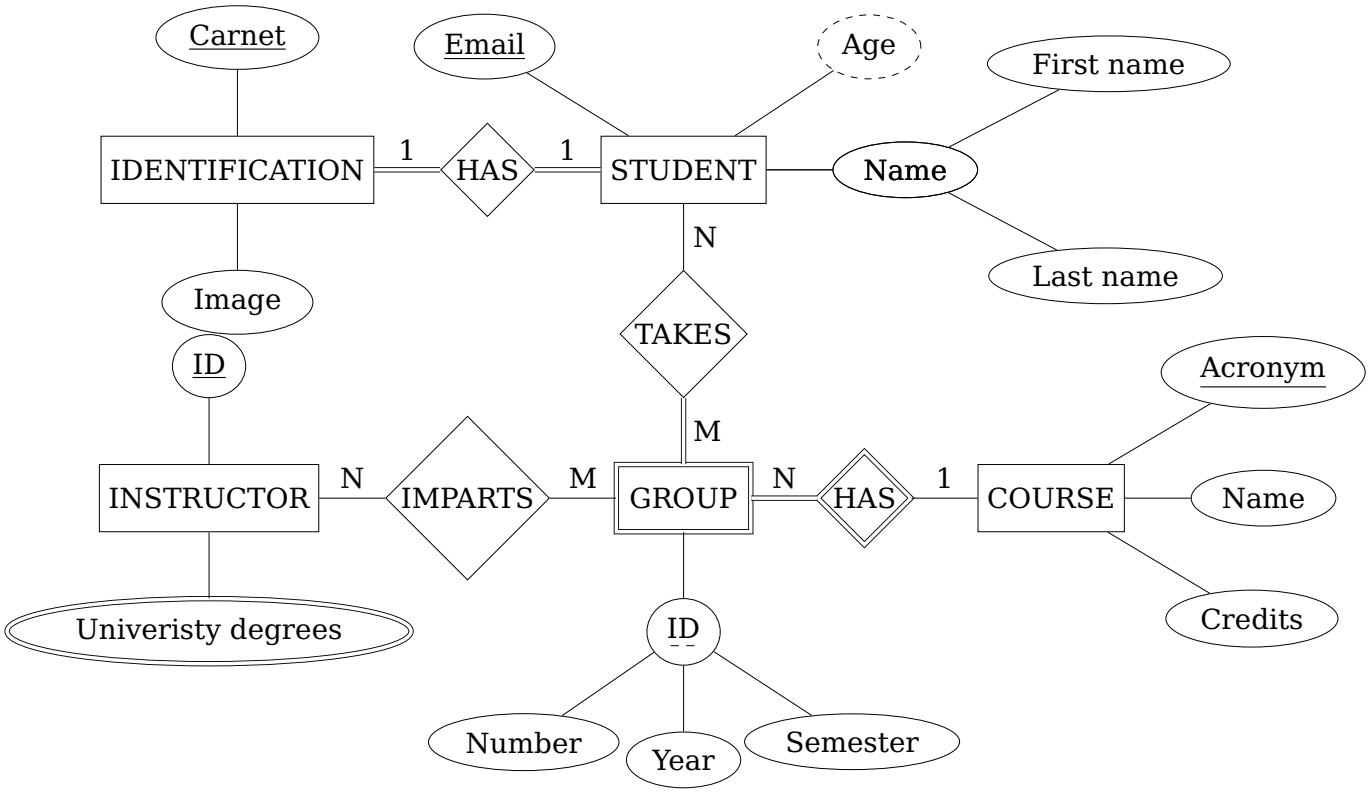


Figura 3.2: UCR student enrollment ER

The following steps are used to map between both models:

Step 1. Create for every strong entity type E a relation R with all the simple attributes of E as attributes of R. For composite attributes, select the simple components. Choose one of the key attributes in E as the key of R, with a composite key in E using all of the simple attributes as the key of R.
Note: We will also create an attribute in the relation for derivative keys. Though, we need SQL to implement the logic of the derivative key.

STUDENT

Email	First_name	Last_name	Age

IDENTIFICATION

Carnet	Image

INSTRUCTOR

ID

COURSE

Acronym	Name	Credits

Figura 3.3: Relations created in step 1

Step 2. Create for every weak entity type W a relation R with all the simple attributes or simple components of composite attributes of W as attributes of R. The key of W is the combination of the primary key of the owner entities and the partial key of the weak entity W.

GROUP

Number	Semester	Year	Acronym

Figura 3.4: Relations created in step 2

Step 3. For every binary 1:1 relationship R, identify the entity types S and T that participate in the relationship. There are three possibilities:

Foreign key approach. For the relation related to the entity type S, that preferible has total participation to reduce NULLs in the database, add a foreign key of T in S. Furthermore, add the simple attributes or components as attributes in S. This is the most common approach used.

IDENTIFICATION

Carnet	Image	Student_email

Figura 3.5: Relations updated in step 3 with the foreign key approach

Merged relation. If both participations between entities are total, we can combine the entity types S and T into a single relation.

STUDENT

Email	First_name	Last_name	Age	Carnet	Image

Figura 3.6: Relations updated in step 3 with the merged relation approach

Cross-reference or relationship relation approach. A third relation U can be created to cross reference the primary keys of S and T as foreign keys.

The primary key of U will be a one of the foreign keys, while the other foreign key will be a unique key (as it is a 1:1 relationship). We save the simple attributes or components inside the new relation U.

HAS

Identification_Carnet	Student_Email

Figura 3.7: Relations updated in step 3 with the cross-reference approach

Step 4. For every binary 1:N relationship R, we apply either the foreign key or the cross-reference approach. The entity S selected to save the foreign key or the primary key, respectively, is the entity at the N-side of the relationship. For example, if GROUP was not weak this would be the entity that would either save the foreign key or have a new table as a primary key.

Step 5. For every binary N:M relationship R, we apply the cross-reference approach. We use both keys of the relations S and T as the primary key.

IMPARTS

ID	Number	Semester	Year	Acronym

TAKES

Email	Number	Semester	Year	Acronym

Figura 3.8: Relations created in step 5

Step 6. For every multivariate attribute, create a new relation R with attributes of the multivariate attribute and a foreign key to the table S with the multivariate attribute.

INSTRUCTOR_UNIVERSITY_DEGREES

ID	University_degree

Figura 3.9: Relations created in step 6

Step 7. For every n-ary relationship above $n > 2$, we follow repeat step 5 but with the primary key being all the keys of all participating entities. The participating entities that have a cardinality of 1 are foreign keys, but not part of the primary key.

The final result of the mapping is shown in Fig. 3.10.

STUDENT

Email	First_name	Last_name	Age	Carnet	Image

INSTRUCTOR

ID

COURSE

Acronym	Name	Credits

GROUP

Number	Semester	Year	Acronym

IMPARTS

ID	Number	Semester	Year	Acronym

TAKES

Email	Number	Semester	Year	Acronym

INSTRUCTOR_UNIVERSITY_DEGREES

ID	University_degree

Figura 3.10: Relational model after mapping ER

3.5. Mapping EER

To map the EER, we can extend the previous steps with an additional step.

Step 8 Choose one of the following mapping schemes, based on the type on the constraints of the subclasses:

8a A relation R is created for the superclass as normal. A relation S_i is created for every subclass of R, with foreign and primary keys, the primary key of superclass R. Every relation S_i also has their respective attributes.

8b A relation S_i is created for every subclass of R, with their primary key as the primary key of superclass R. Every relation S_i also has the attributes of superclass R and their respective attributes.

8c A relation R is created for the superclass. The attributes of the subclass S_i of R are saved in R. Furthermore, R saves a type attribute to represent the different subclasses.

8d A relation R is created for the superclass. The attributes of the subclass S_i of R are saved in R. Furthermore, R saves a bool type attribute to represent if the relation belongs to that subclass.

The relationship between the constraints and the type of mapping scheme is shown in Table 7.1.

	Disjoint	Overlapping
8A	✓	✓
8B	✓	✗
8C	✓	✗
8D	✓	✓

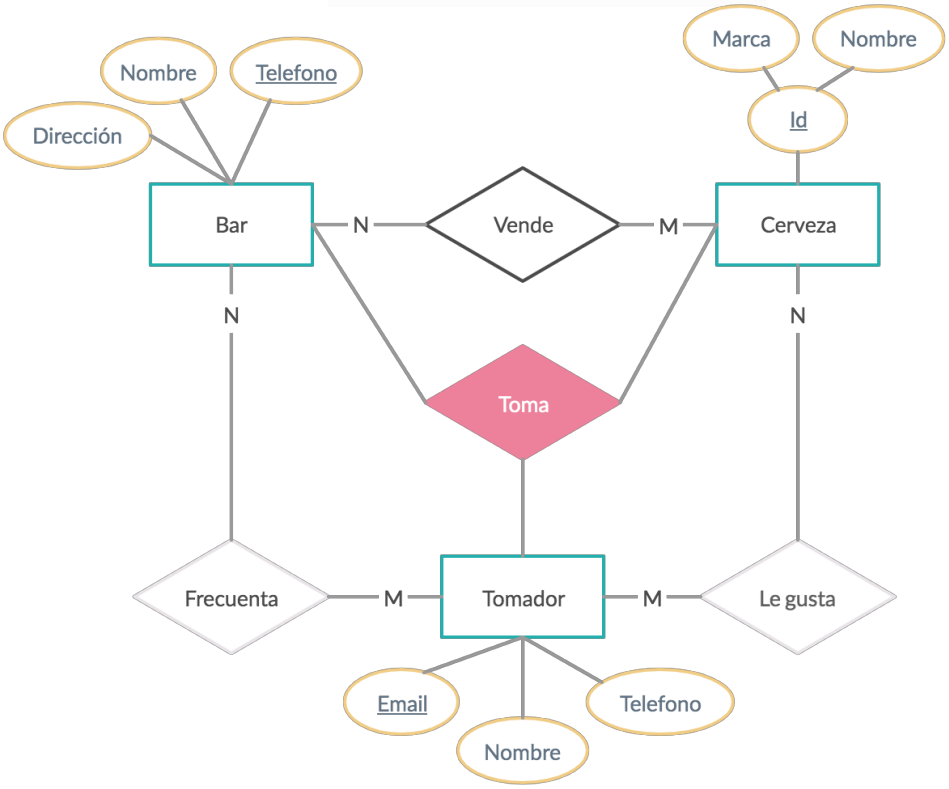
Figura 3.11: Relationship between the mapping scheme of the subclasses with the constraints

3.6. Review

Simples

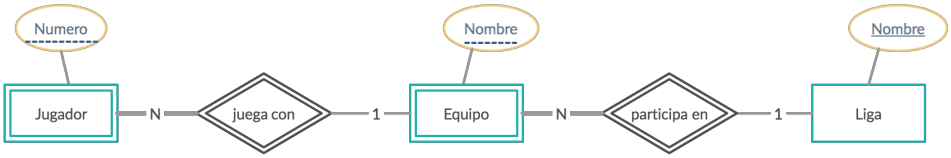
Sistema de bares

De: Dra. Alexandra Martínez



Sistema de ligas

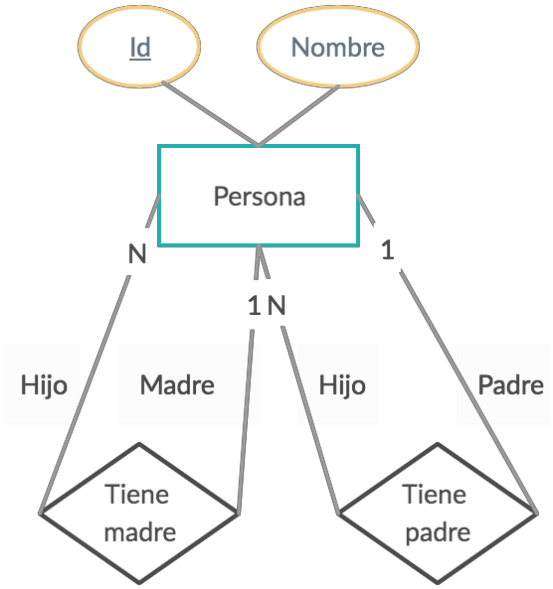
De: Dra. Alexandra Martínez



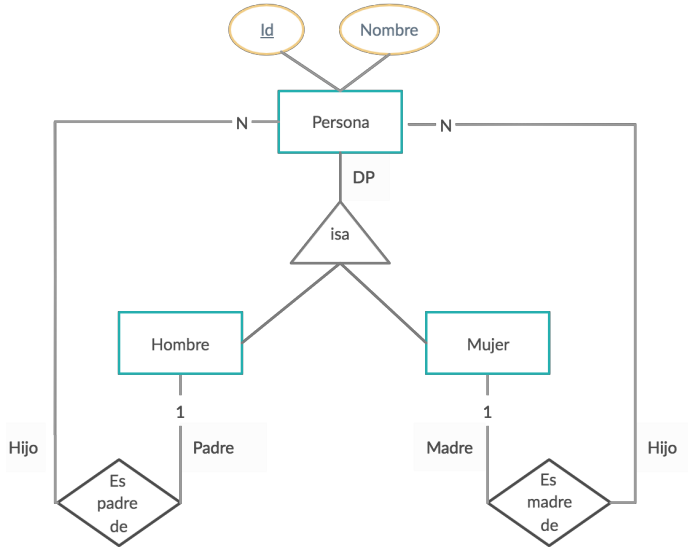
Sistema de genealogía

De: Dra. Alexandra Martínez

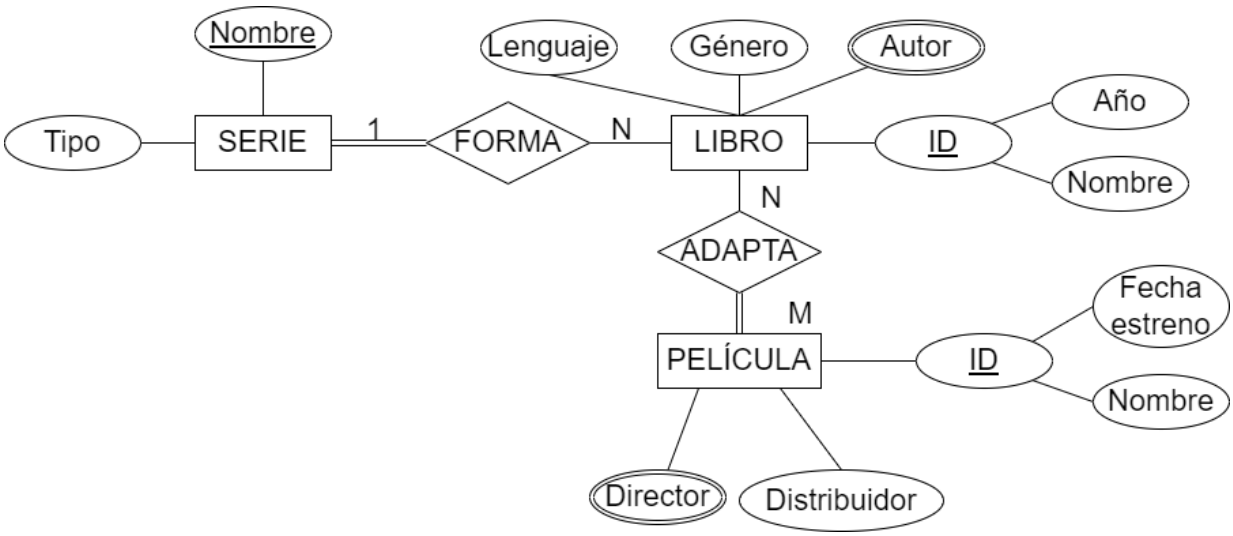
Parte 1:



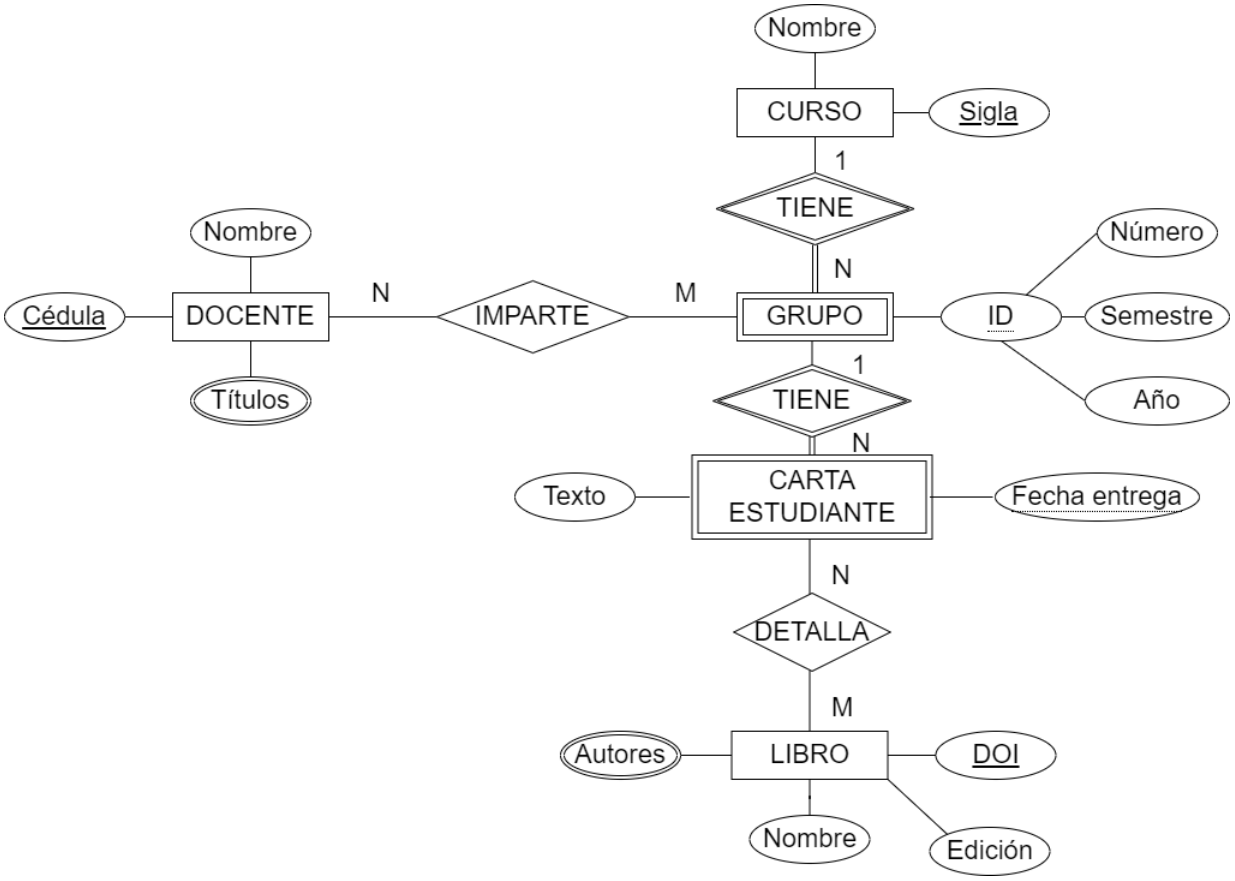
Parte 2:



Sistema de adaptaciones



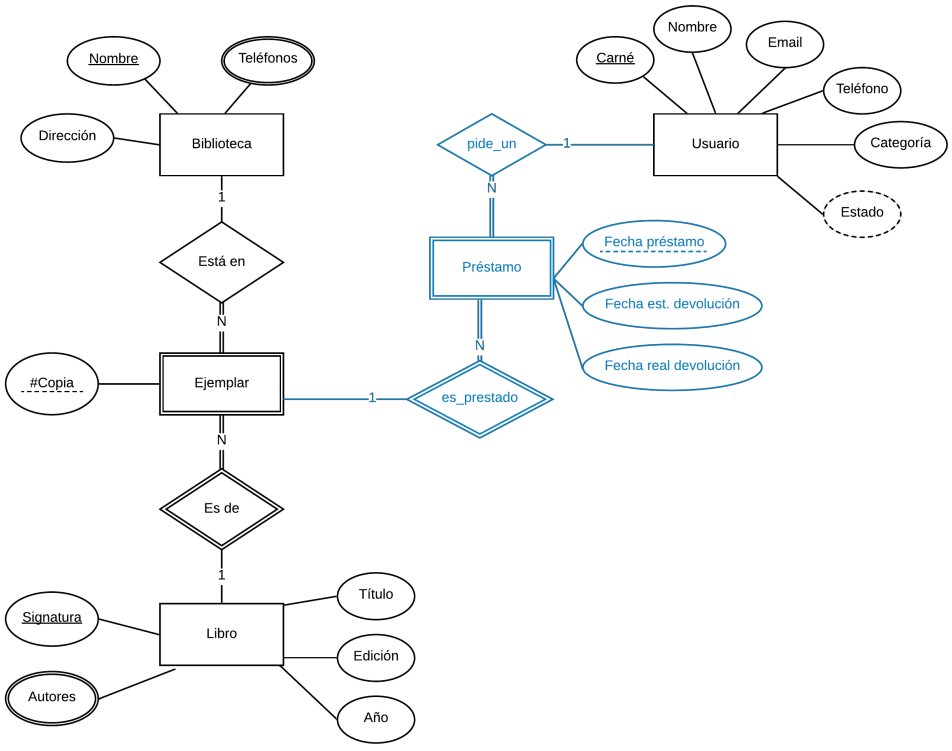
Sistema de cartas al estudiante



Mediano

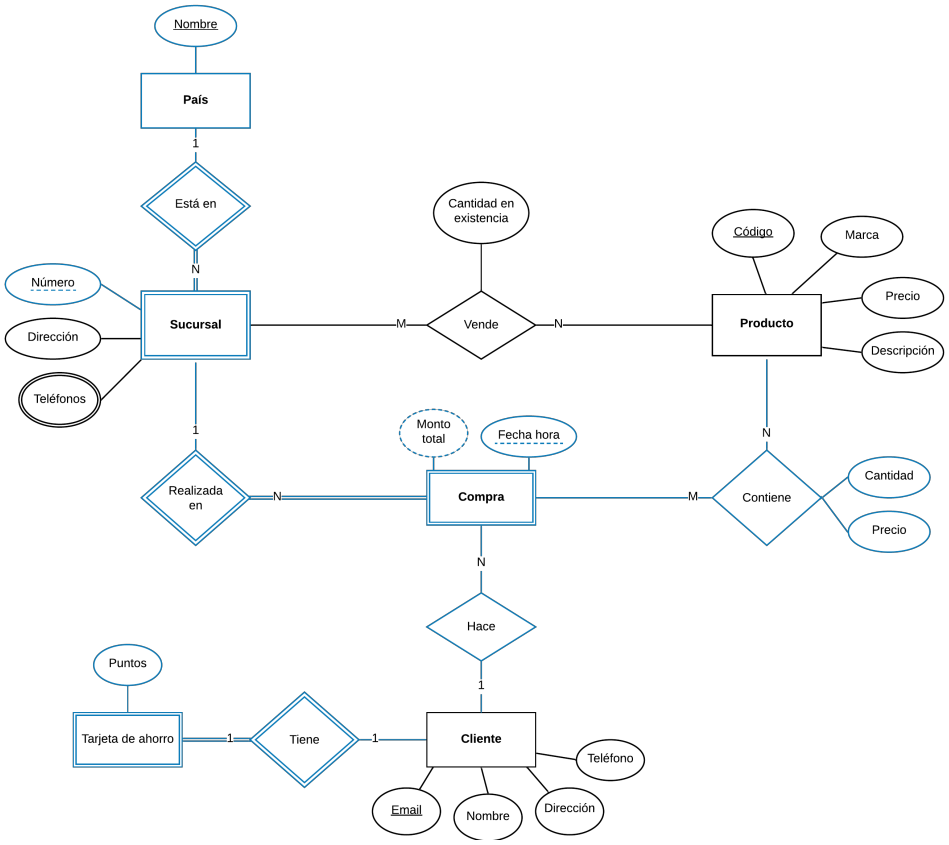
Sistema de bibliotecas BIBS

De: Dra. Alexandra Martínez



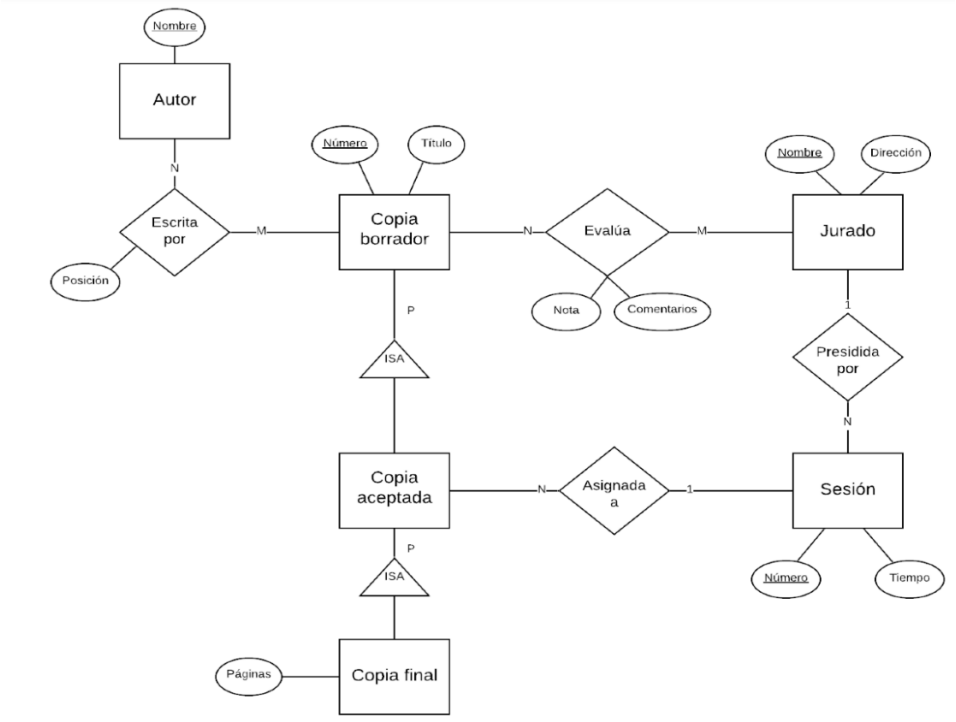
Sistema de supermercados SUPERK

De: Dra. Alexandra Martínez



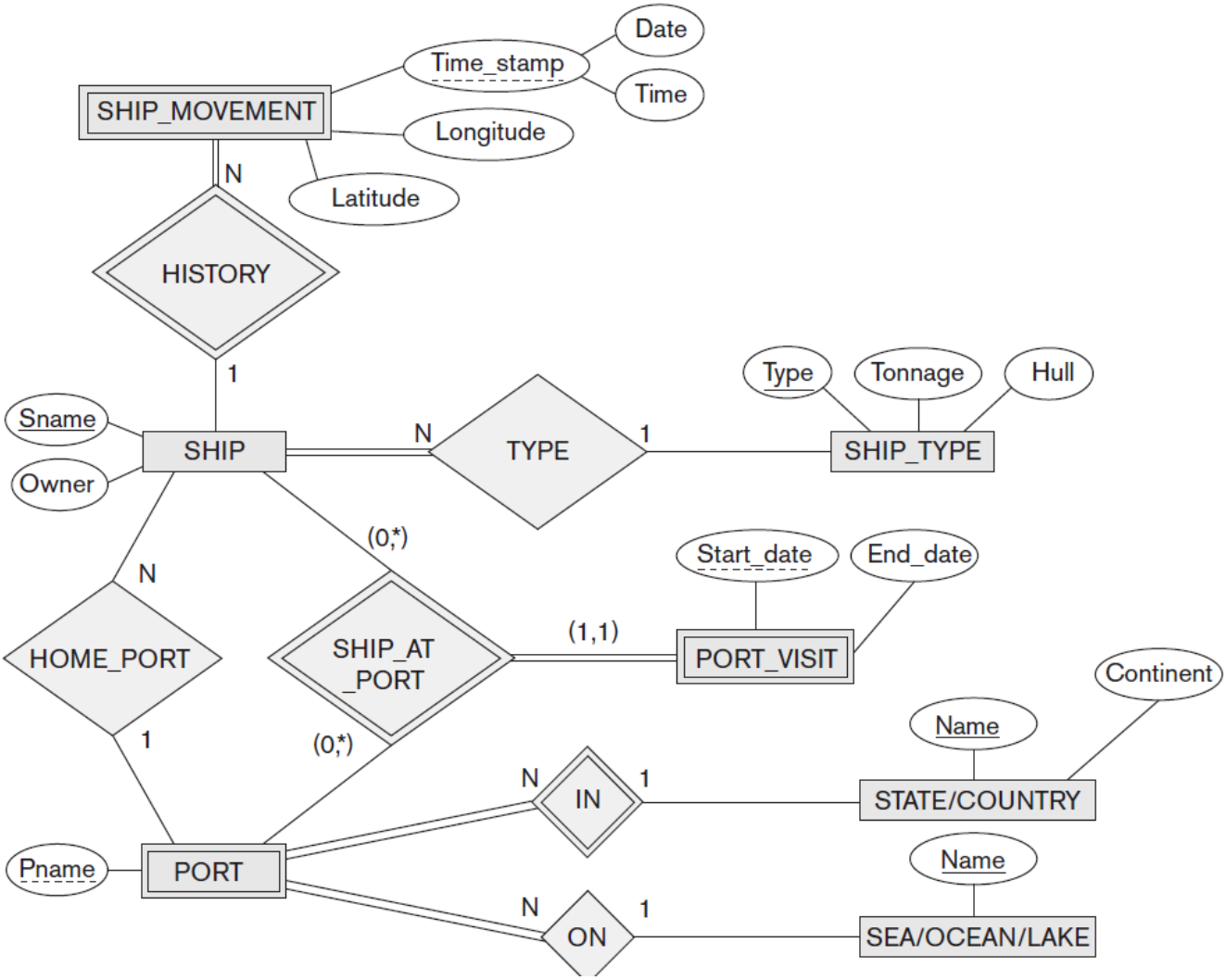
Sistema de Conferencia

De: Dra. Alexandra Martínez



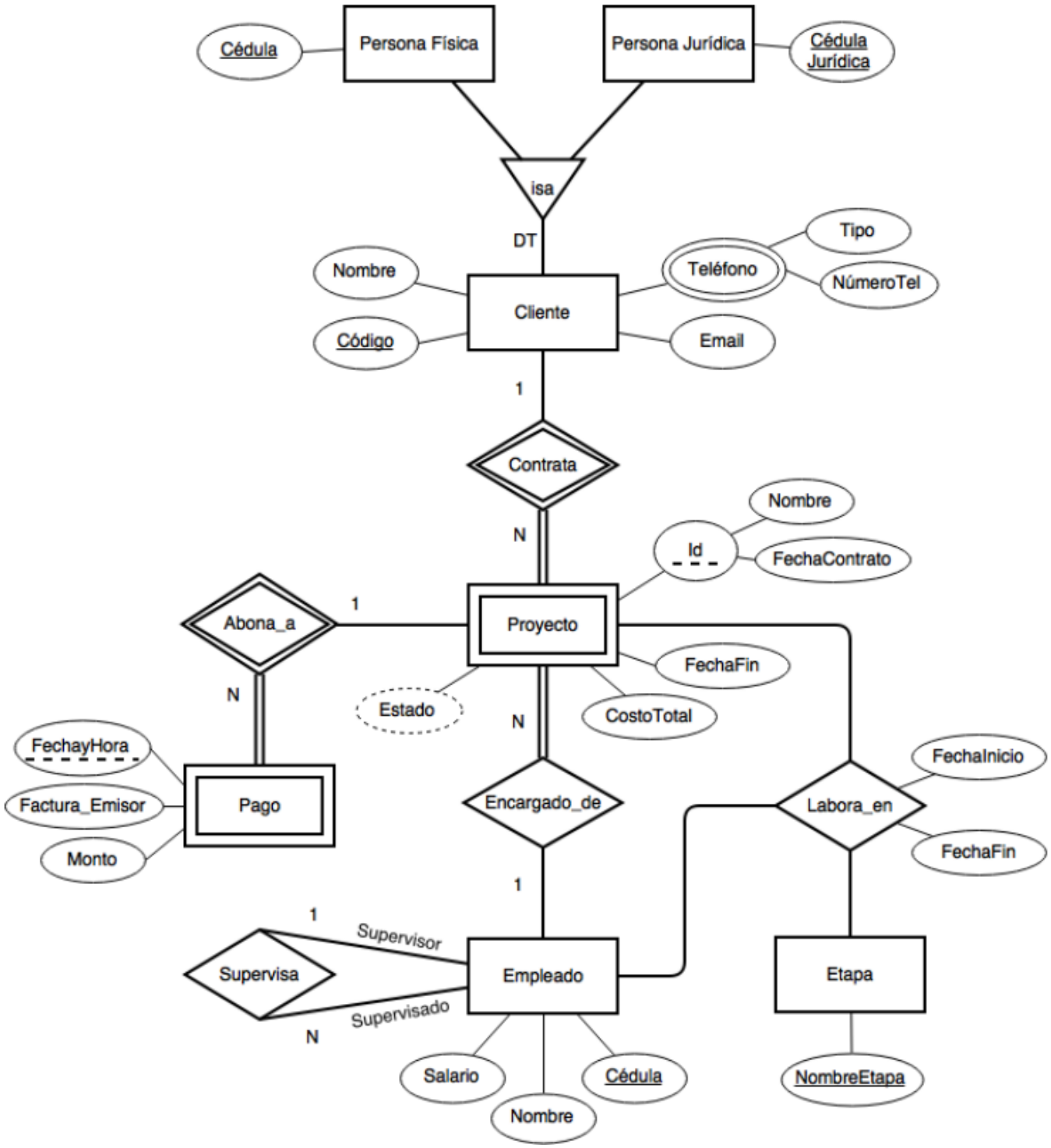
Sistema de Buques

De: R. Elmasri and S. Navathe, Fundamentals of database systems, 7th ed.



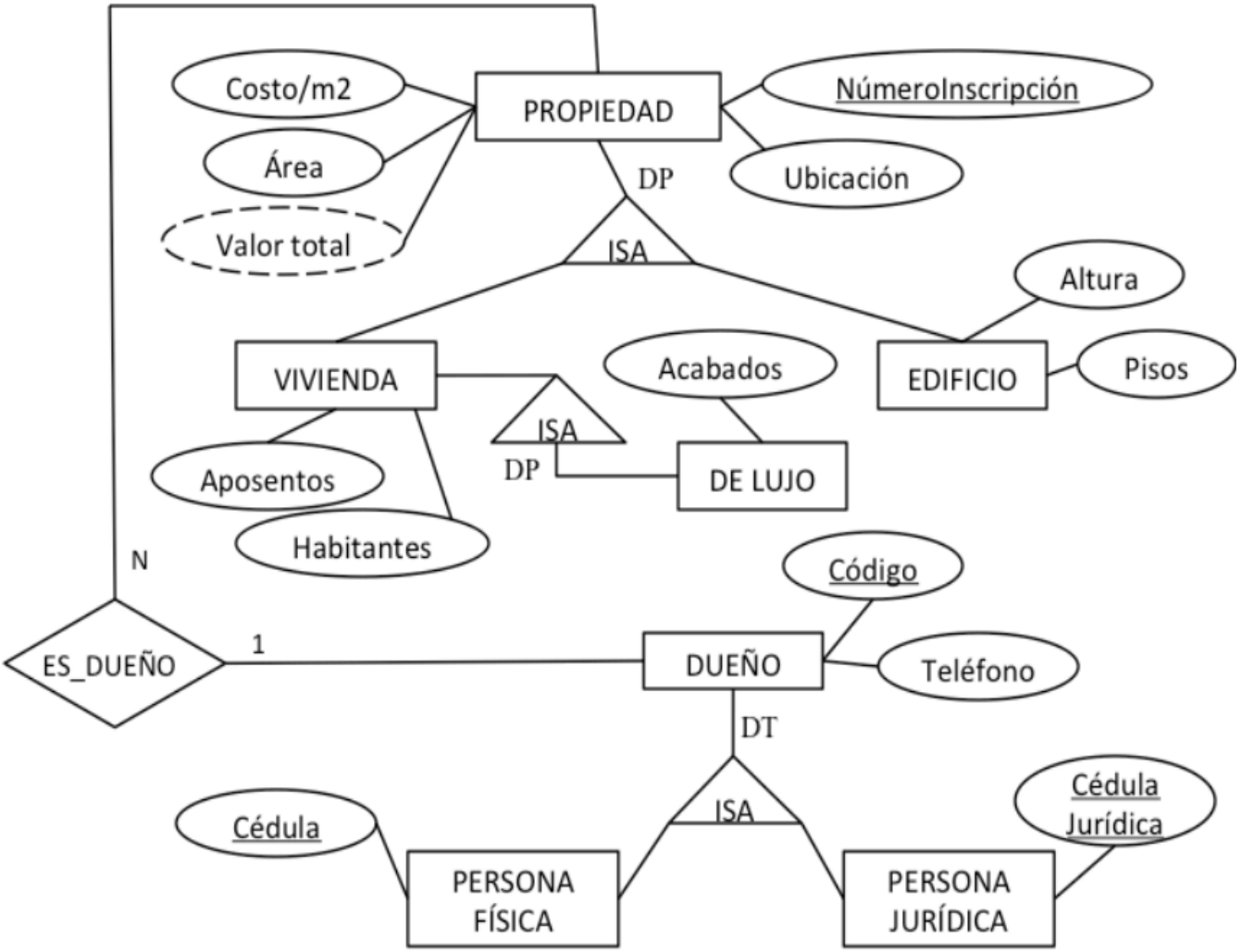
Sistema de Proyectos

De: Dra. Alexandra Martínez

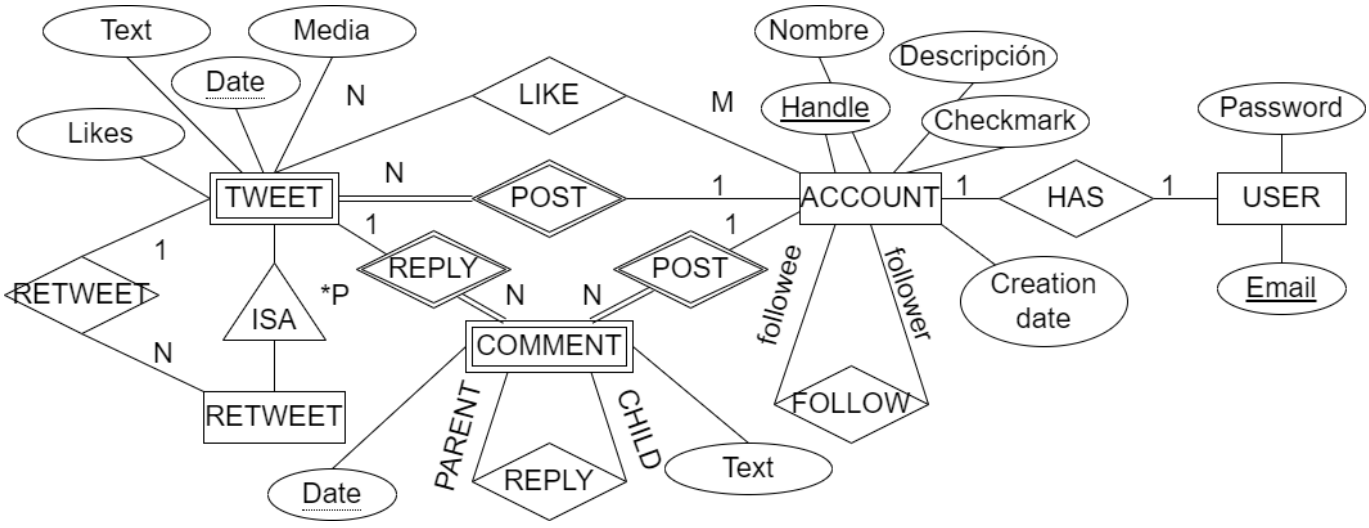


Sistema de Propiedades

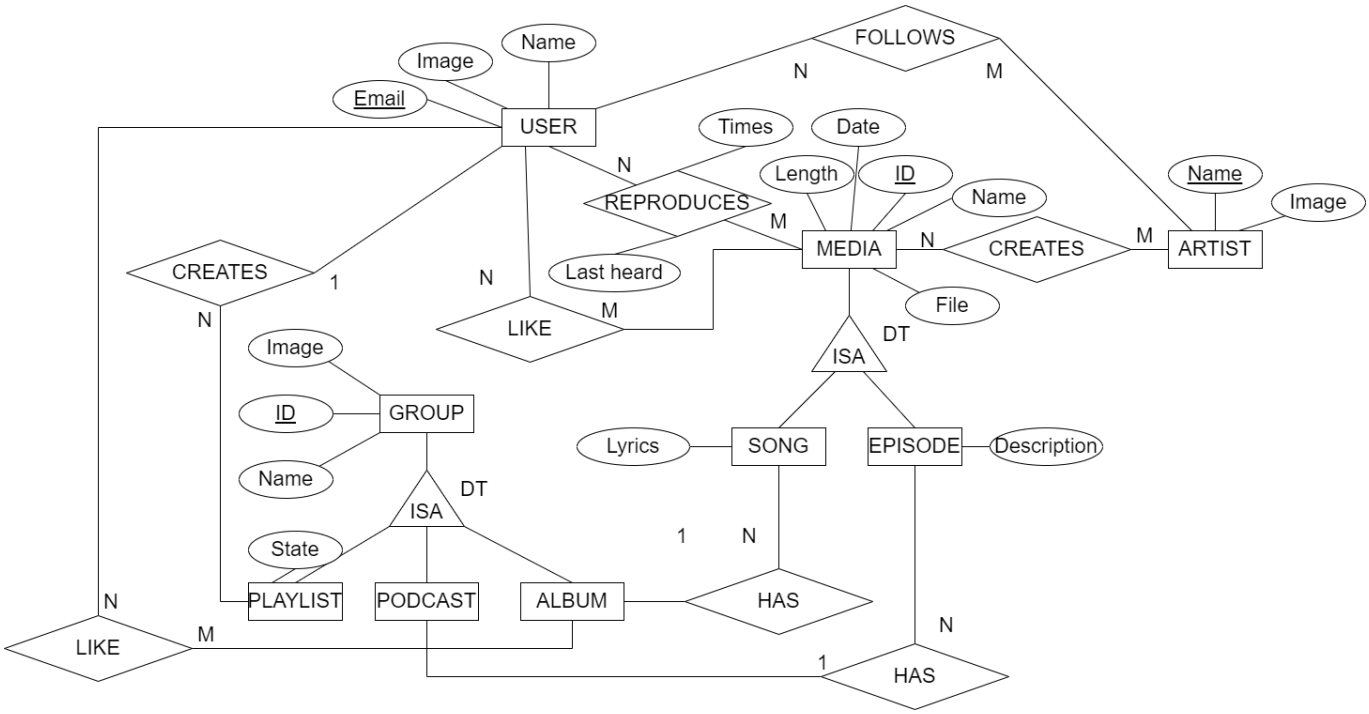
De: Dra. Alexandra Martínez



Sistema de Twitter

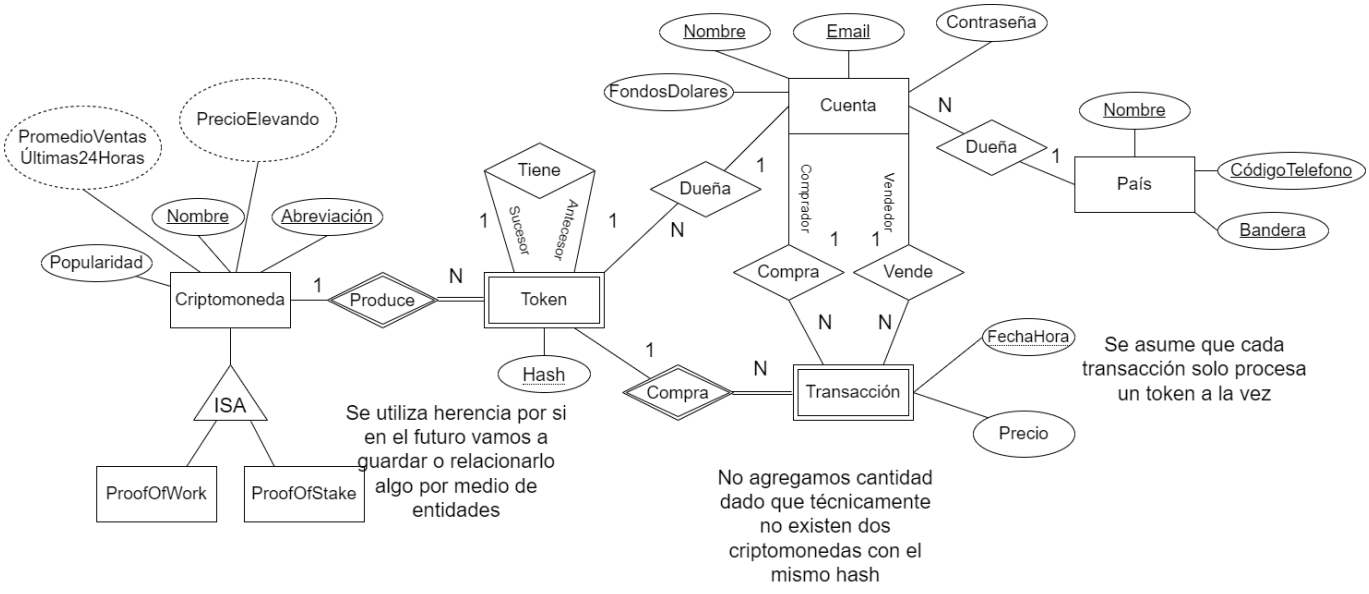


Sistema de Spotify

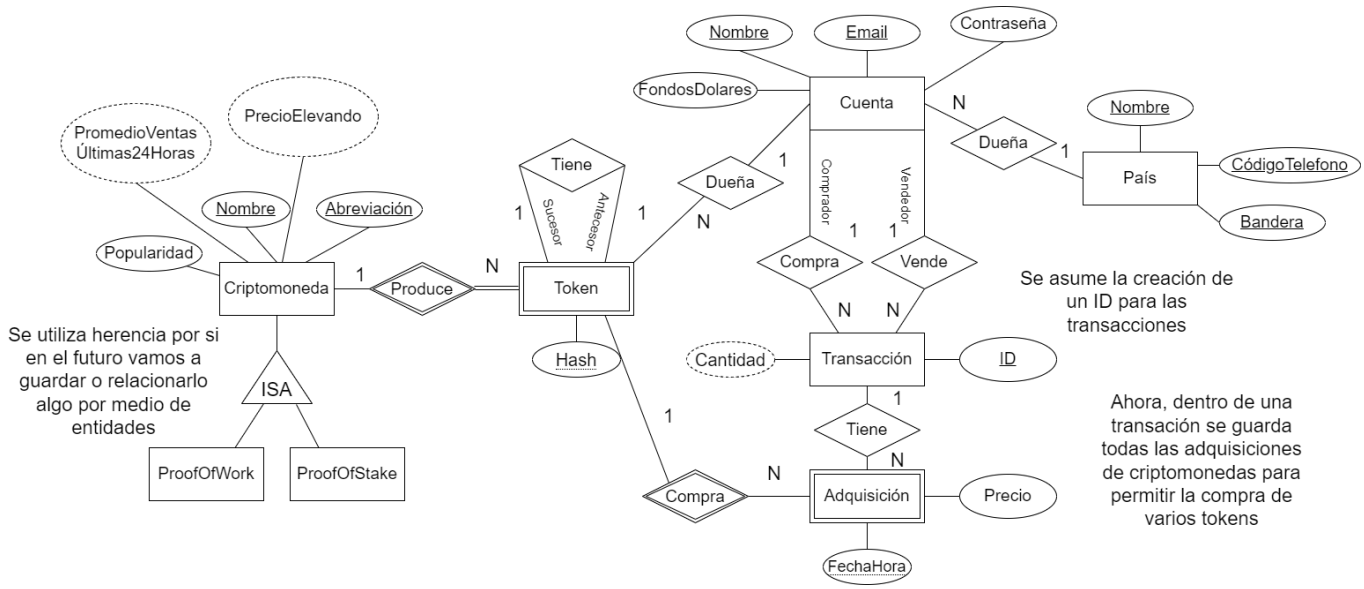


Sistema de Criptomonedas

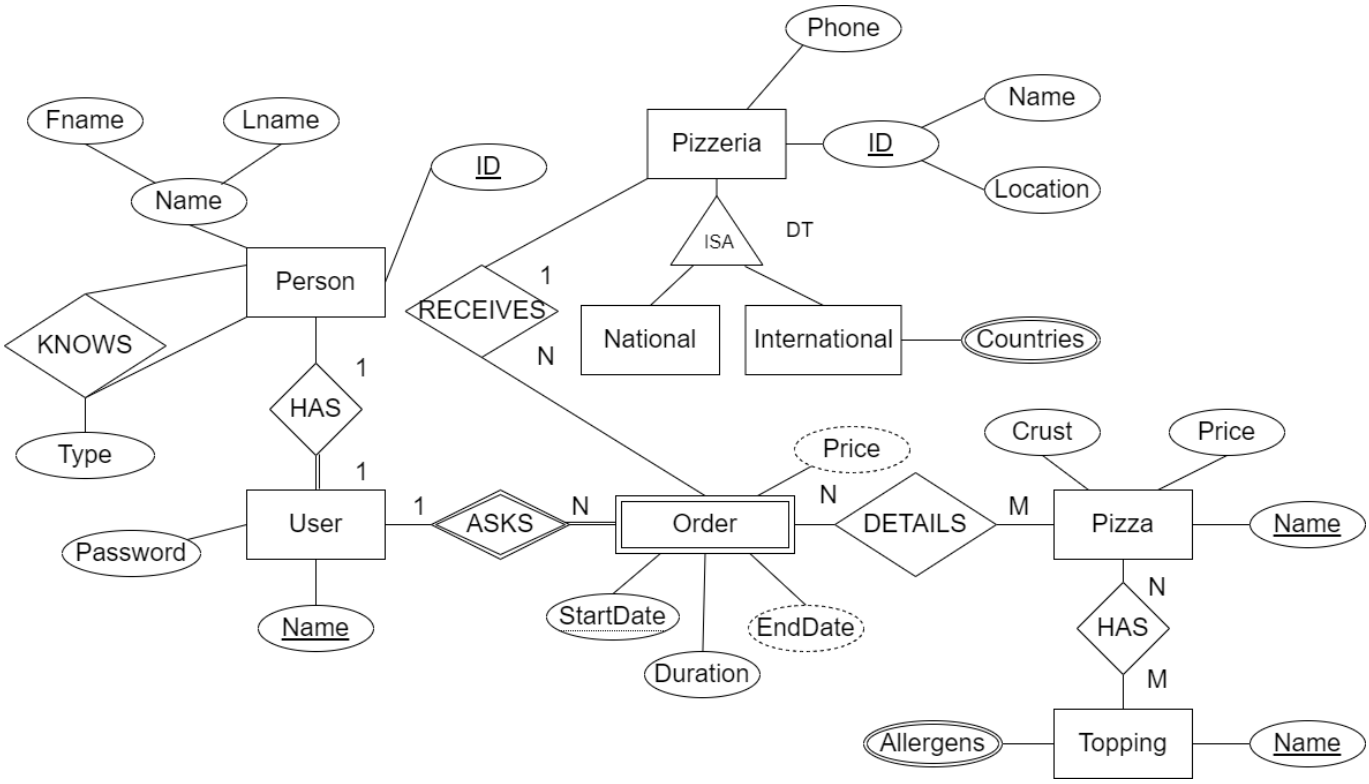
Versión 1:



Versión 2:

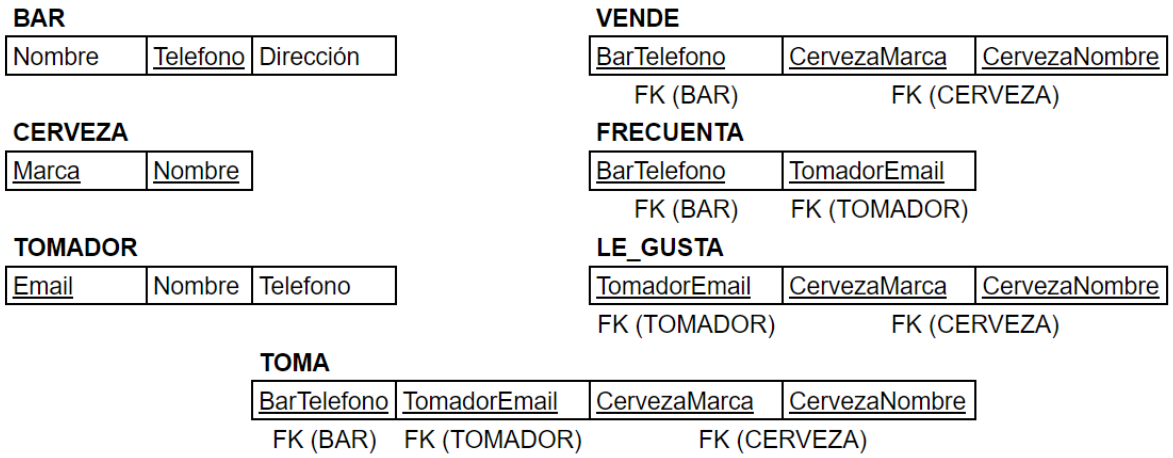


Sistema de Pizza

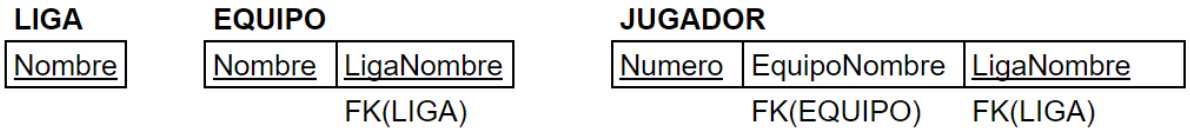


3.7. Solutions

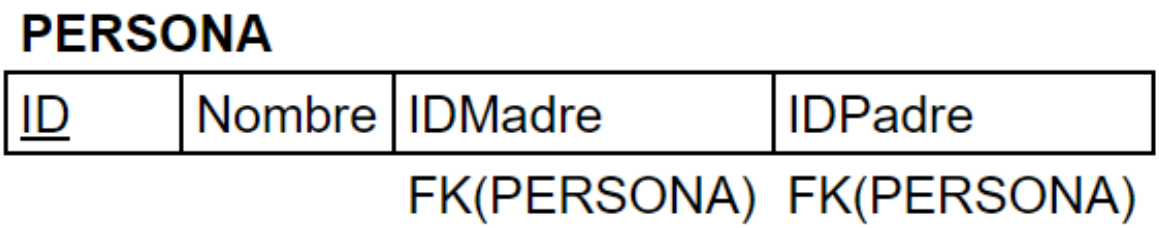
Sistema de bares



Sistema de ligas

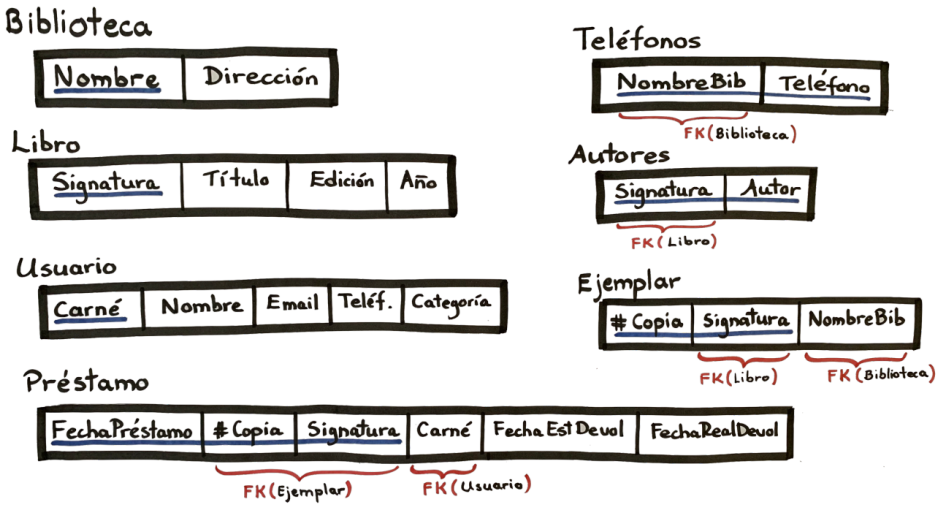


Sistema de genealogía



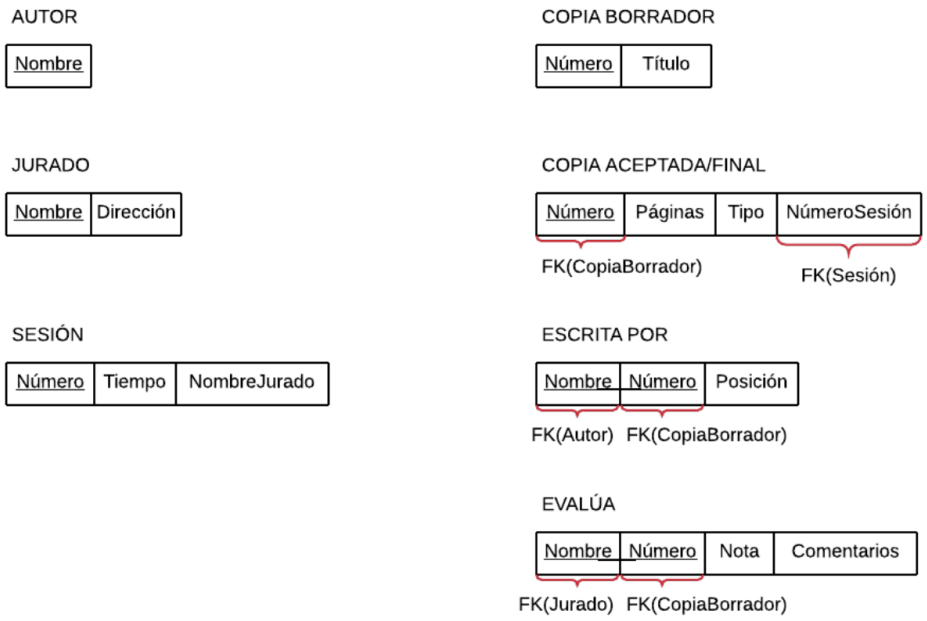
Sistema de bibliotecas BIBS

De: Dra. Alexandra Martínez



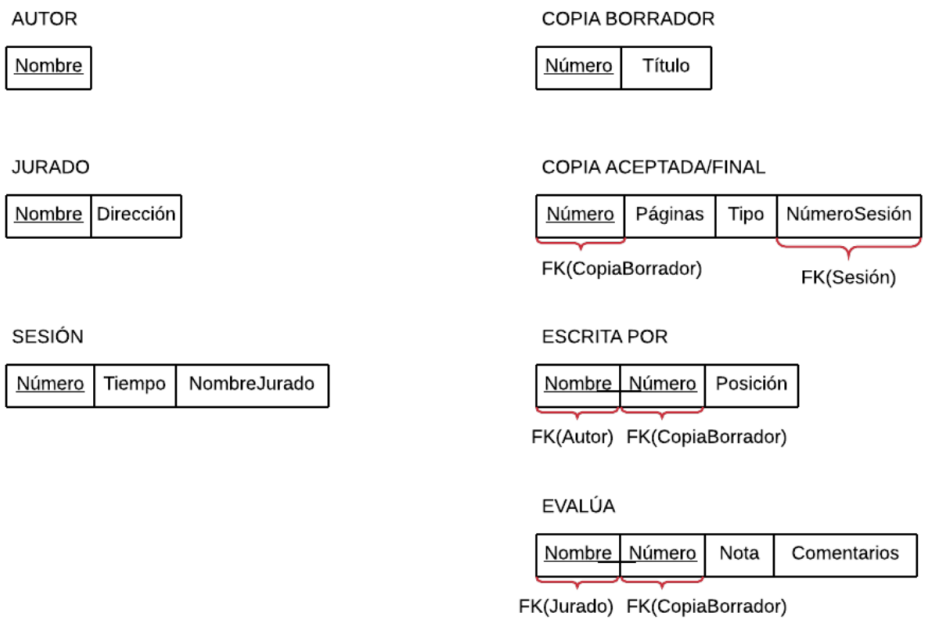
Sistema de Conferencia

De: Dra. Alexandra Martínez



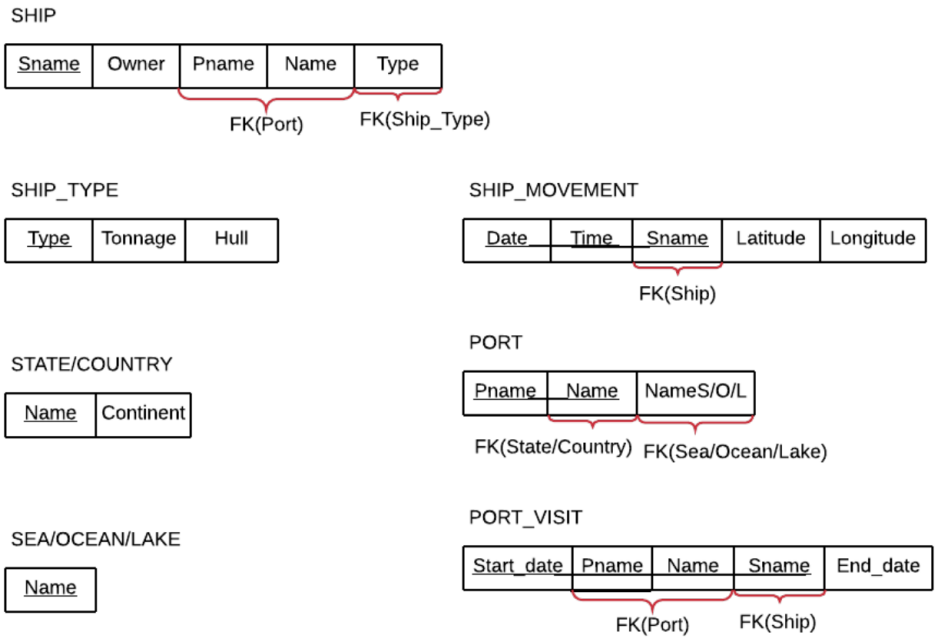
Sistema de Conferencia

De: Dra. Alexandra Martínez



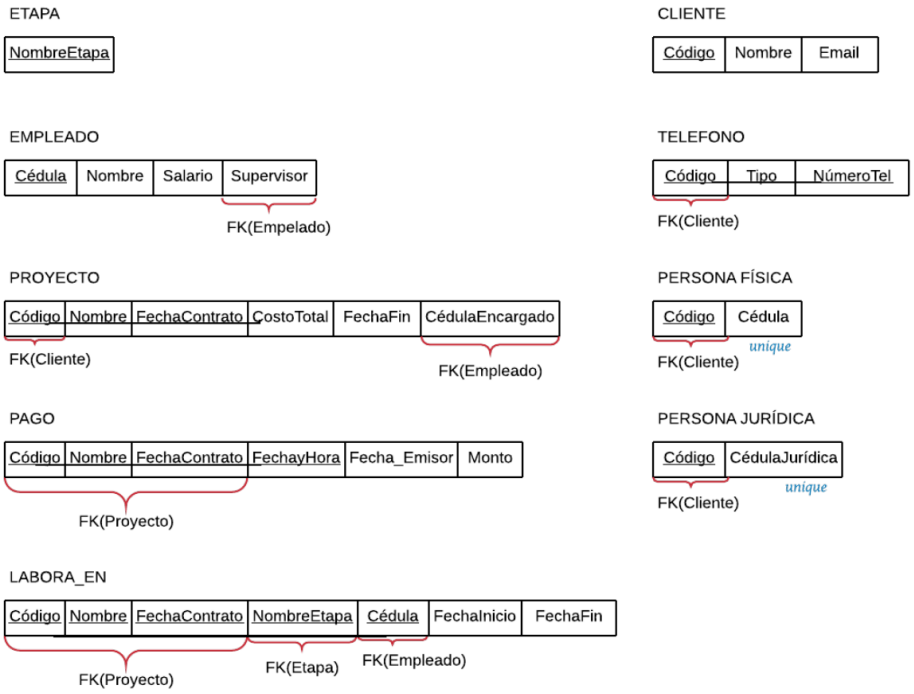
Sistema de Buques

De: Dra. Alexandra Martínez



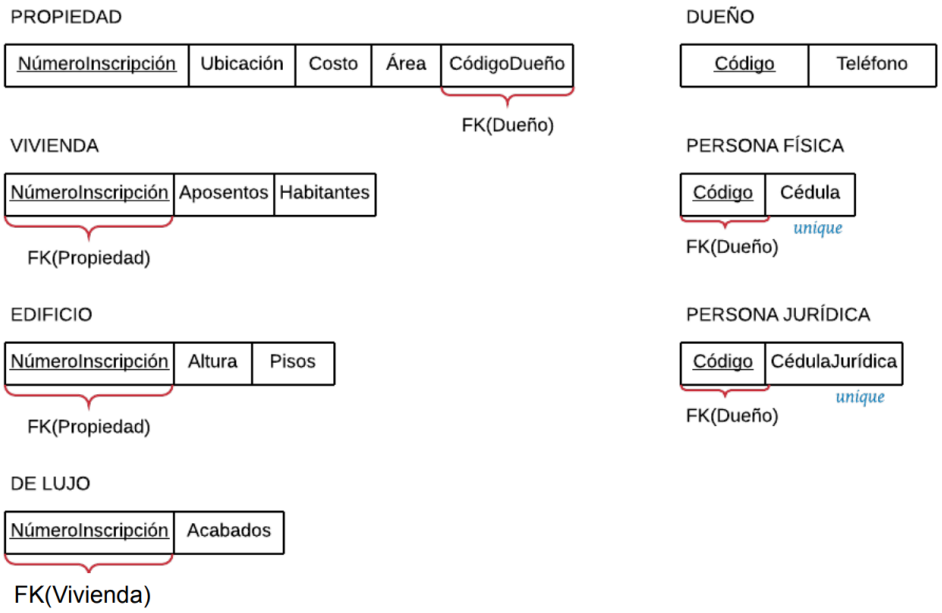
Sistema de Proyectos

De: Dra. Alexandra Martínez



Sistema de Propiedades

De: Dra. Alexandra Martínez



Sistema de Pizza

Person

ID	Lname	Fname

Knows

PersonIDKnowee	PersonIDKnoweer	Type
FK(Person)	FK(Person)	

User

Password	Name	PersonID
		FK(Person)

Pizzeria

Name	Location	Phone

PizzeriaNational

PizzeriaName	PizzeriaLocation
FK(Pizzeria)	

PizzeriaInternational

PizzeriaName	PizzeriaLocation
FK(Pizzeria)	

PizzeriaInternationalCountries

PizzeriaName	PizzeriaLocation	Country
FK(PizzeriaInternational)		

Topping

Name

ToppingAllergens

ToppingName	Allergens
FK(Topping)	

Pizza

Name	Crust	Price

Order

UserName	StartDate	Duration	EndDate	Price	PizzeriaName	PizzeriaLocation
FK(User)					FK(Pizzeria)	

OrderDetailsPizza

UserName	StartDate	PizzaName
FK(Order)		FK(Pizza)

PizzaHasTopping

PizzaName	ToppingName
FK(Pizza)	FK(Topping)

Se escoge mapeo 8A dado que:

- Se puede dado que es DT el tipo de herencia
- Es extensible en el futuro si se quieren agregar nuevos atributos o relaciones al modelo.
- No desperdicia espacio en nulos

Parte III

Implementación

Capítulo 4

SQL Básico

4.1. Structured Query Language

The Structured Query Language (SQL) represents a practical relational model for relational DBMS (RDBMS). As there are slight differences with how different RDBMS implement SQL. SQL Server Management Studio (SSMS) as it is the IDE for SQL Server (our RDBMS).

- It defines *what* the results are, not *how* to get them.
- It is based on *relational calculus*.
- It is not the standard language for RDBMS.
- It was defined as a standard by the American National Standards Institute (ANSI) and the International Standard Organization (ISO).
- It defines a language for both DDL and DML.

4.2. Data definition

The relationship between the terms of the relational model and SQL are shown in Table 4.1.

Relational model	SQL
Relation	Table
Tuple	Row
Attribute	Column

Cuadro 4.1: Relationship between the terms of the relational model and SQL

Within SQL, there are different structures that represent the database, as shown in Fig. 4.1. In the following subsections, each of these levels are detailed.

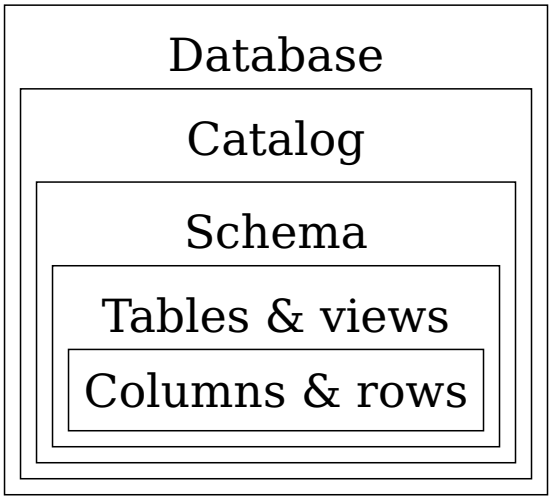


Figura 4.1: Abstraction levels of database concepts

Database

We can **CREATE** an instance of a database [1] with:

```
CREATE DATABASE DB_SIVANA;
```

To use that database, we use:

```
USE DB_SIVANA;
```

We can also add the authorized users with:

```
CREATE DATABASE DB_SIVANA AUTHORIZATION 'sivana';
```

Within SSMS, we can see all the databases within the *Object Explorer* > *Databases*, as shown in Fig. 4.2.

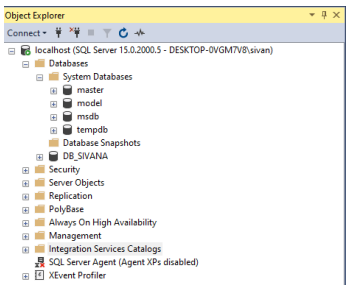


Figura 4.2: Databases in SSMS

[1] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-database-transact-sql

Catalog

A catalog is a collection of schemas [2]. All the schemas within the catalog are saved in the *INFORMATION_SCHEMA*. We can find this schema in *Database* > *Security* > *Schemas* (Fig. 4.3).

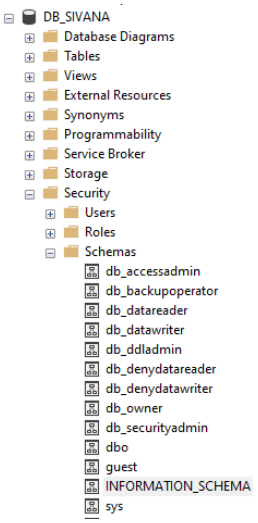


Figura 4.3: INFORMATION_SCHEMA in SSMS

Schema

An SQL Schema represents a group of tables or views that belong to the same database application. It has a:

- Name: Way to identify the Schema
- Authorization identifier: Indicates the user who owns the schema.
- Descriptors: To define the elements within the schema.

We can **CREATE** the schema [3] with:

```
CREATE SCHEMA UNIVERSITY;
```

Or we can also add who is authorized by:

```
CREATE SCHEMA UNIVERSITY AUTHORIZATION 'sivana';
```

We can later add the descriptors by **ALTER**ing the schema [4] and delete the schema by **DROP**ping it [5]. We can determine the current schema with:

```
SELECT SCHEMA_NAME();
```

When we log in into the database installation, we automatically connect to the default catalog and schema. We can change the default schema with:

[2] https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/catalog-views-transact-sql
[3] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-schema-transact-sql
[4] https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-schema-transact-sql
[5] https://docs.microsoft.com/en-us/sql/t-sql/statements/drop-schema-transact-sql

```
ALTER USER 'sivana' WITH DEFAULT_SCHEMA=UNIVERSITY;
```

We can find the schemas in *Database* > *Security* > *Schemas* (Fig. 4.4).

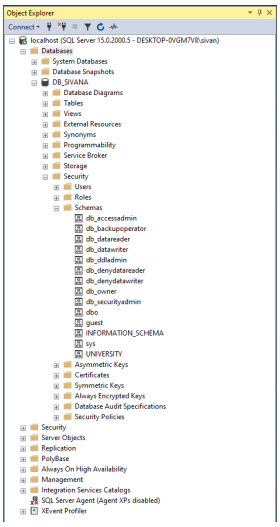


Figura 4.4: Schemas in SSMS

Table with columns

A table defines a relation with its name, attributes and initial constraints (Not NULL, keys [6], uniqueness and checks [7]). We can **CREATE** [8] the table with:

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6) PRIMARY KEY,
    NAME VARCHAR(50) NOT NULL,
    CREDITS INT NOT NULL,
    UNIQUE (NAME)
);
```

Implicitly, as we are within the default schema (dbo), while we create the table it executes the query as if it was:

```
CREATE TABLE dbo.COURSE;
```

We can see the result of creating the table in *Database* > *Tables* (Fig. 4.5). There are different types of tables for SSMS [9]. If we expand a table, we can also see its columns, keys, constraints, triggers, indexes and statistics. Double clicking on any of these elements will show more details for each of them.

[6] https://docs.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints

[7] https://docs.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints

[8] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql

[9] https://docs.microsoft.com/en-us/sql/relational-databases/tables/tables

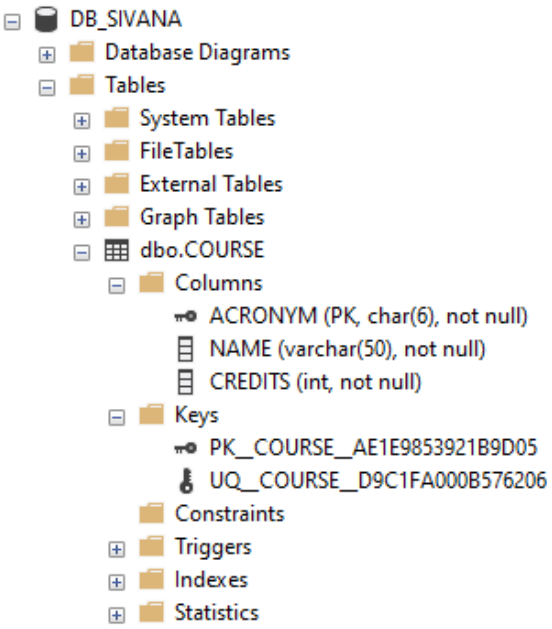


Figura 4.5: Schemas in SSMS

We can CREATE a foreign key as:

```
CREATE TABLE _GROUP (
    ACRONYM CHAR(6),
    NUMBER INT,
    SEMESTER INT,
    YEAR INT,
    PRIMARY KEY (ACRONYM, NUMBER, SEMESTER, YEAR),
    FOREIGN KEY (ACRONYM) REFERENCES COURSE(ACRONYM)
);
```

To define a foreign key, the table must already exist. Therefore, if there is a foreign key fk_1 for table T_1 that depends on a foreign key fk_2 for table T_2 such that fk_1 references T_2 and fk_2 references T_1. Then, the restriction can be defined by **ALTER**ing [10] the table. If we define a column as a primary key, it will automatically add the *NOT NULL* constraint.

4.3. Data types

The following data types can be created, as shown in Table 4.2.
Some details with regards to the data types:

- Strings are case sensitive.
- Strings are declared with apostrophes (e.g., ‘Sivana’).
- We can concatenate strings with || (e.g., ‘gallo ’ || ‘pinto’).
- We declare bits with a B preceeding the digits within apostrophes (e.g., B‘101’).
- For dates, we can also use timezones with TIME WITH TIME ZONE. Without it, the default is the local time zone for the SQL session.

[10] `https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-table-transact-sql`

Category	Data type	Description	Types
Numeric	Integers	An integer number	INT or INTEGER SMALLINT
	Real	A real number	FLOAT or REAL DOUBLE PRECISION
	Formated	Formated number, where i is the number of decimal digits and j the decimal digits after the decimal point	DECIMAL/DEC/NUMERIC(i, j)
Chars	Fixed length	Fixes a char string to have n chars. If shorter, it is padded with spaces to the right.	CHAR(n)
	Variable length	Strings can have a variable length of maximum n chars	VARCHAR(n)
	Large object	Specifies large texts like documents, where n is a size and s the size type (K,M,G)	CLOB(ns)
Bits	Fixed length	Fixes a bit string to have n bits.	CHAR(n)
	Variable length	Strings can have a variable length of maximum n bits.	BIT VARYING(n)
	Large object	Specifies large bit string, where n is a size and s the size type (K,M,G)	BLOB(ns)
	Boolean	Boolean that can be either TRUE or FALSE	BIT
Dates	Date	Given in 'YYYY-MM-DD'.	DATE
	Time	Given in 'HH-MM-SS'.	TIME

Cuadro 4.2: SQL data types

- Some implementations of SQL include more or less types [11] (e.g., SQL does not have an explicit BOOLEAN data type).

4.4. Constraints

We can define several constraints while creating tables. In the following subsections, different commands are explained.

NOT NULL

For an attribute, we can define a **NOT NULL** constraint to not allow NULL values. This is always implicitly declared for the primary key of the relation. We can see an example of a **NOT NULL** constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6) PRIMARY
KEY,
    NAME VARCHAR(50) NOT NULL,
    CREDITS INT NOT NULL,
    ...
);
```

We can see that we can define for the columns *NAME* and *CREDITS* the constraint after specifying the name and data type. For the primary key, it is not necessary to specify the constraint as it is done implicitly. We can see within SSMS the restriction for the columns with the constraint in *Database* > *Tables* > *Table* > *Columns* > *Column*, as shown in Fig. 4.6.

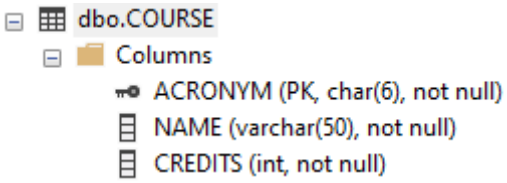


Figura 4.6: NOT NULL constraint in SSMS

DEFAULT

A default value can be set for a column, with the **DEFAULT** command. If no default is set for a column, the default is *NULL* for all the columns without the **NOT NULL** constraint. We can see an example of this constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ...
    CREDITS INT NOT NULL DEFAULT 4,
    ...
);
```

[11] https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql

We set that the *CREDITS* column will have as a default value a four. We can see within SSMS the restriction for the columns with the constraint in *Database* > *Tables* > *Table* > *Constraints*, as shown in Fig. 4.7.

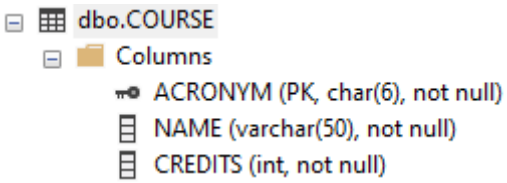


Figura 4.7: DEFAULT constraint in SSMS

CHECK

We can use a **CHECK** constraint to define a domain, row-based and general constraints. We can see an example of a domain constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ...
    CREDITS INT NOT NULL DEFAULT 4,
    CHECK (CREDITS > 0 AND CREDITS < 13),
    ...
);
```

With this constraint, a *CREDITS* for a *COURSE* has to be $0 < CREDITS < 13$. We can see within SSMS the restriction in *Database* > *Tables* > *Table* > *Constraints*, as shown in Fig. 4.8.

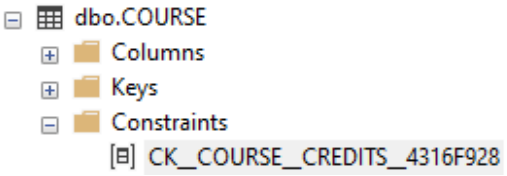


Figura 4.8: CHECK constraint in SSMS

The constraint can also be defined in the following form:

```
CREATE DOMAIN CREDITS AS INT
CHECK (CREDITS > 0 AND CREDITS < 13);
```

We can also define a tuple constraint, where each row that is inserted or updated is checked individually. For example, if we add for **_GROUP** columns for the number of spaces available (*NUMBER_AVAILABLE_SPACES*) and the number of enrolled students (*NUMBER_ENROLLED*). Thus, we can define a row-based constraint to not allow more students enrolled than the available spaces as follows:

```
CHECK (NUMBER_ENROLLED <= NUMBER_AVAILABLE_SPACES);
```

PRIMARY KEY

We can define a **PRIMARY KEY** constraint while creating the column for the table. We can see an example of this constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6) PRIMARY KEY,
    ...
);
```

We can also define a primary key of several columns after defining the columns, as seen in the following example for *GROUP*.

```
CREATE TABLE _GROUP (
    ACRONYM CHAR(6),
    NUMBER INT,
    SEMESTER INT,
    YEAR INT,
    PRIMARY KEY (ACRONYM, NUMBER, SEMESTER, YEAR),
    ...
);
```

We can see within SSMS the restriction in *Table* > *Columns* and *Table* > *Keys*, as shown in Fig. 4.9.

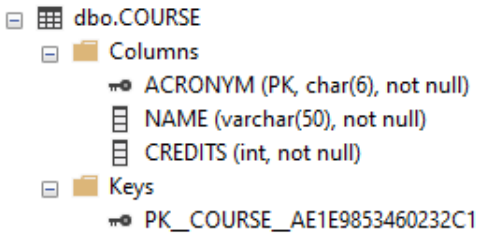


Figura 4.9: PRIMARY KEY constraint in SSMS

UNIQUE

The **UNIQUE** clause defines candidate keys that are unique for the table. We can see an example of this constraint for *COURSE*.

```
CREATE TABLE COURSE (
    ...
    NAME VARCHAR(50) NOT NULL UNIQUE,
    ...
);
```

A **UNIQUE** constraint can also be defined for a group of columns. This can be defined with the following structure, putting the list of columns inside the parentheses:

```
CREATE TABLE COURSE (
    NAME VARCHAR(50) NOT NULL,
    UNIQUE (NAME),
    ...
);
```

We can see within SSMS the restriction in *Table* > *Keys*, as shown in Fig. 4.10.

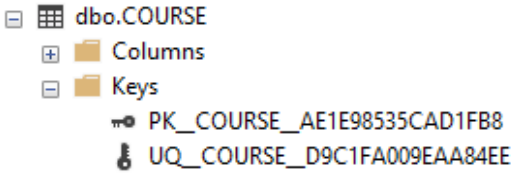


Figura 4.10: UNIQUE constraint in SSMS

FOREIGN KEY

A **FOREIGN KEY** constraint can be defined for a table. This constraint ensures that the constraint is followed while tuples are added or deleted, or when a foreign key or primary key is updated. We can see an example of this constraint for *GROUP*.

```
CREATE TABLE _GROUP (
    ACRONYM CHAR(6),
    FOREIGN KEY (ACRONYM) REFERENCES COURSE(ACRONYM)
    ...
);
```

We can see within SSMS the restriction in *Table* > *Columns* and *Table* > *Keys*, as shown in Fig. 4.11.

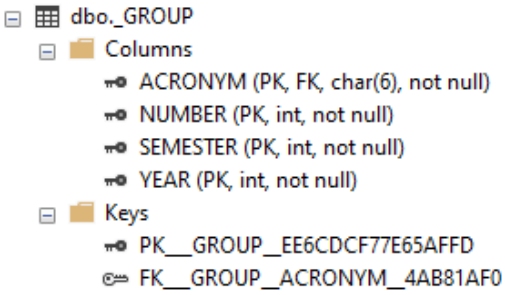


Figura 4.11: FOREIGN KEY constraint in SSMS

We can define different types of foreign key constraints. The *referential trigger actions* defines the behavior during changes to the foreign key:

- **NO ACTION:** An error is raised and the state of the database is rolled back. This is also known as the **RESTRICT** option and is the default behavior.
- **CASCADE:** When a parent table updates or deletes a foreign key, the reference table *updates* the foreign key. This does not work if the referenced or foreign key is a **TIMESTAMP**.

- **SET NULL:** When a parent table updates or deletes a foreign key, the reference table *sets to NULL* the foreign key.
- **SET DEFAULT:** When a parent table updates or deletes a foreign key, the reference table *sets to the default values* the foreign key. For this constraint to work, all columns of the foreign key must have a default. If a column is nullable without an explicit default value, the default value is NULL.

These actions can be combined for different **FOREIGN KEY** restrictions. Furthermore, restrictions can fire secondary cascading chains and can be subsequently repeated. If the **SET NULL**, **SET DEFAULT** or **CASCADE** action triggers a table with **NO ACTION**, all the updates are stopped and rolled back. These behaviors can be defined for updates (**ON UPDATE**) or deletes (**ON DELETE**). We can see how this works in the following example:

```
CREATE TABLE _GROUP (
    ...
    FOREIGN KEY (ACRONYM) REFERENCES COUR-
SE(ACRONYM)
        ON DELETE NO ACTION ON UPDATE NO ACTION,
    ...
);
```

CONSTRAINT

For every constraint, we can define a name by specifying the keyword **CONSTRAINT**, then the name of the constraint and then the constraint to be named. We can see an example for the *COURSE* table.

```
CREATE TABLE COURSE (
    ACRONYM CHAR(6),
    CONSTRAINT
PK_ACRONYM_COURSE
        PRIMARY KEY(ACRONYM),
    ...
);
```

We can see within SSMS the constraint with the name, for this example, in *Table > Keys*, as shown in Fig. 4.12.

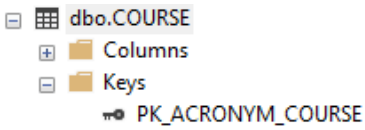


Figura 4.12: Naming a PRIMARY KEY constraint in SSMS

4.5. Data modification

There are three ways we can modify the database: inserts, deletes and updates. In the following subsections, each subtype is described and defined.

INSERT

We can **INSERT** [12] into a table rows. It is possible to bulk insert rows [13]. The block of commands are the following:

```
INSERT INTO <table (<attribute list>)>
VALUES <value list>;
```

- **INSERT INTO:** Details the table that rows will be inserted. A list of attributes can be given inside the parentheses to define the order of the columns, but is optional.
- **VALUES:** Details the values that will be inserted. The order is given either implicitly (defined by the order in which the table was defined) or explicitly (defined by the order given in the attribute list).

The first way to insert is to detail the table and the values for the columns, in the same way it was defined. For example, here we can see the result of inserting a row into *COURSE*.

```
INSERT INTO COURSE
VALUES ('CI0127', 'Bases de Datos', 4);
```

We can also **INSERT** by explicitly mentioning the order of the columns while inserting. It is necessary to specify all the columns that cannot be NULL and have no default. If a column can be NULL or has a default and is not specified, it will automatically add a NULL or the default value, respectively. This explicit **INSERT** is shown in the following example:

```
INSERT INTO COURSE (NAME, CREDITS, ACRONYM)
VALUES ('Ingeniería de Software', 4, 'CI0126');
```

DELETE

We can **DELETE** [14] delete current rows of a table. This may activate constraints. The full block of commands are the following:

```
DELETE FROM <table>
WHERE <condition>
```

- **DELETE FROM:** Defines the table that rows are deleted from.
- **WHERE:** Details a boolean condition that identifies the rows that will be deleted. This clause is optional.

We can delete all the rows in a table with the following command:

[12] https://docs.microsoft.com/en-us/sql/t-sql/statements/insert-transact-sql
[13] https://docs.microsoft.com/en-us/sql/t-sql/statements/bulk-insert-transact-sql
[14] https://docs.microsoft.com/en-us/sql/t-sql/statements/delete-transact-sql

```
DELETE FROM COURSE;
```

DELETing rows does not remove the table definition in the database. To delete a table, we must **DROP** it as follows:

```
DROP COURSE;
```

To **DELETE** only certain rows from a table, we can define a *condition* using the **WHERE** clause.

```
DELETE FROM COURSE
WHERE CREDITS > 5 ;
```

UPDATE

We can also **UPDATE** [15] column values for rows. This command is similar to **DELETE**, but has a **SET** clause that defines the values of the updated columns. The full block of commands are the following:

```
UPDATE <table>
SEMESTER <list of columns with values>
WHERE <condition>
```

- **UPDATE:** Defines the table that rows are updated.
- **SET:** Details the list of columns with the values to update them.
- **WHERE:** Details a boolean condition that identifies the rows that will be updated. This clause is optional.

The following is an example that updates all rows in the table:

```
UPDATE COURSE
SET CREDITS = 5;
```

We also define that only rows that meets a *condition* will be updated with the **WHERE** clause, as follows:

```
UPDATE COURSE
SET CREDITS = 10
WHERE ACRONYM = 'CI0127';
```

We can also modify the current value of a column, using the **SET** clause as follows:

```
UPDATE COURSE
SET CREDITS = CREDITS + 1;
```

[15] https://docs.microsoft.com/en-us/sql/t-sql/queries/update-transact-sql

4.6. Data retrieval

To view the rows that are in the database, we can use the **SELECT** [16] command. This command has multiple clauses, with the only required clauses being **SELECT** and **FROM**. The block of commands are the following (we will see some more in advanced SQL):

```
SELECT <attribute list>
FROM <table list>
WHERE <condition>
ORDER BY <attribute list>;
```

- **SELECT:** Details the list of columns that will be retrieved from the tables. All the columns can be retrieved with *.
- **FROM:** Details all the tables needed for the query.
- **WHERE:** Details a boolean condition that identifies the rows that will be retrieved. This clause is optional.
- **ORDER BY:** Defines the order of the rows retrieved. This clause is optional.

We can define a simple query to select all the rows from a table that meet a condition. This is shown in the following example where all the *COURSE*s' *ACRONYM*s for *COURSE*s with more than three *CREDITS* are gathered. To compare values within the **WHERE** clause we use comparison operators. The comparison operators grammar for SQL and the respective math representation are shown in Table 4.3.

```
SELECT ACRONYM
FROM COURSE
WHERE CREDITS > 3;
```

Math	SQL
$=$	=
$\neq$	<>
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=

Cuadro 4.3: Comparison operators relationship between math and SQL

We can also define a **SELECT** without a **WHERE** clause to select all the rows in a table as shown in the following example. Here, we gather all the *ACRONYM*s for every row in *COURSE*.

```
SELECT ACRONYM
FROM COURSE;
```

[16] https://docs.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql

We desire to get all the columns that we can after an operation. One way to do this is to specify all the columns manually. We could also use * in the **SELECT** clause, specifying that all the columns will be retrieved. In the following example, the attributes *ACRONYM*, *NAME* and *CREDITS* are gathered for all the rows in *COURSE* that have more than three credits.

```
SELECT *
FROM COURSE
WHERE CREDITS > 3;
```

A range of values in the **WHERE** condition can be defined with a **BETWEEN**. In the following example, only *COURSE*s with *CREDITS* between three and seven are retrieved.

```
SELECT *
FROM COURSE
WHERE (CREDITS BETWEEN 3 AND 7);
```

In the **FROM** clause we can put more than one table as a list of tables. In the following example, we retrieve the *combinations* of *COURSE*s and *GROUP*s, gathering the *NAME*, *CREDITS*, *NUMBER*, *SEMESTER* and *YEAR*. The combinations retrieved have no relationship if the tables reference one another, they are all the possible combinations between both tables.

```
SELECT NAME, CREDITS, NUMBER, SEMESTER, YEAR
FROM COURSE, _GROUP
```

Implicitly, if there are no attributes that have the same name within the list of tables it treats the column as if it said “Table.Column”. In our current example query, an implicit column is *NAME* that is treated as *COURSE.NAME*. If there are two columns with the same name in the list of tables, we must specify *explicitly* this reference.

If we want to specify the tables based on a condition between both tables, we can do it with a *join condition*. Thus, in the **WHERE** clause we specify the column in a table that relates with another table. In the following example we can see the join condition for “C.ACRONYM = G.ACRONYM”. Note that the keyword **AS** creates an alias for the tables that can be used to explicitly state the column.

```
SELECT NAME, CREDITS, NUMBER
FROM COURSE AS C, _GROUP AS G
WHERE C.ACRONYM = G.ACRONYM;
```

In a **WHERE** clause we can combine various conditions with **AND**s and **OR**s, following boolean logic. There is also a **NOT** condition. We can see in the following example how we can specify the *join condition*, the *GROUP*’s semester must be equal to one and the *GROUP*’s year must be equal to 2022.

```
SELECT NAME, CREDITS, NUMBER
FROM COURSE AS C, _GROUP AS G
WHERE SEMESTER = 1 AND YEAR = 2022 AND C.ACRONYM = G.ACRONYM;
```

This query can be further refined to specify the order of the results with the **ORDER BY** clause. This will order the rows starting from the leftmost element of the

list of columns to the rightmost (i.e., order first by the first column, second by the second column and so forth). We can further define the order with **ASC** (ascending order) and **DESC** (descending order). The default order is **ASC**. An example is shown in the following query, where the rows are ordered in ascending order by year and then semester.

```
SELECT NAME, CREDITS, NUMBER
FROM COURSE AS C, _GROUP AS G
WHERE SEMESTER = 1 AND YEAR = 2022 AND C.ACRONYM = G.ACRONYM
ORDER BY YEAR, SEMESTER;
```

When we retrieve values from columns, we are getting all the values from all the rows, but not the unique values within the current database state. Thus, to gather the distinct values in the **SELECT** clause the **DISTINCT** keyword is stated before the column name. In the following example, we can see how we can gather the distinct *YEAR* values in the rows of the *GROUP*s.

```
SELECT DISTINCT YEAR
FROM _GROUP;
```

Character string columns can be pattern matched. We can use the % and **LIKE** keywords to match an arbitrary number of zero or more characters. In the following example, only the *COURSE*s with *ACRONYM*s that start with 'CI' (computer science courses) are matched.

```
SELECT *
FROM COURSE
WHERE ACRONYM LIKE 'CI%';
```

We can also specify the specific number of characters to match with an underscore (_) for each character. In the following example, only the *COURSE*s with *ACRONYM*s that start with 'CI' (computer science courses) and proceeded by four blank characters are retrieved.

```
SELECT ACRONYM
FROM _GROUP
WHERE ACRONYM LIKE 'CI____';
```

SET operations can be used with **SELECT** queries. The operations that are defined in SQL are:

- **UNION:** For sets A and B, this operation includes all the elements in either A or B are included in the resulting set.
- **INTERSECTION:** For sets A and B, this operation includes all the elements that are in both A and B are included in the resulting set.
- **EXCEPT:** For sets A and B, this operation includes all the elements that are A without any element that is shared with B.

If we define these operations as they are, they will generate the rows without duplicates. If we add the keyword **ALL** (e.g., **UNION ALL**) duplicates are not eliminated. The tables must be *type-compatible relations*, thys must have all the same attributes in the same order. In the following example, the *GROUP*s that were imparted in the *SEMESTER* 1 and the *YEAR* 2022 are retrieved (this example can also be done with an AND).

```
(SELECT DISTINCT ACRONYM, NUMBER
FROM _GROUP
WHERE SEMESTER = 1)
INTERSECTION
(SELECT DISTINCT ACRONYM, NUMBER
FROM _GROUP
WHERE YEAR = 2022);
```

4.7. Database permissions

While using a database, different permissions are defined for different users. These permissions can grant access to certain capabilities and data. We can **GRANT** [17] or **REVOKE** [18] permisions for users, based on their respective commands.

[17] https://docs.microsoft.com/en-us/sql/t-sql/statements/grant-transact-sql
[18] https://docs.microsoft.com/en-us/sql/t-sql/statements/revoke-transact-sql

Capítulo 5

SQL Avanzado

5.1. NULL

In SQL, if the column does not have a *NOT NULL* constraint, we can have a *NULL* value.

- A NULL value can mean that the value is unknown, value withheld or it is not applicable. However, we do not know the type of NULL as we save the same NULL value for all cases.
- As we could compare NULL values, SQL uses a three-valued logic (TRUE, FALSE or UNKNOWN) instead of boolean logic (TRUE or FALSE). Thus, the used truth tables in SQL are shown in Tables 5.1, 5.2 and 5.3.

	TRUE	**FALSE**	**UNKNOWN**
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Figura 5.1: AND truth table

	TRUE	**FALSE**	**UNKNOWN**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Figura 5.2: OR truth table

TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Figura 5.3: NOT truth table

- While **SELECT**ing tuples in a **WHERE** clause, only tuples that are TRUE are selected. There is an exception for outer joins.
- To compare nulls we use the **IS** command. The following example shows how it is used, to find all the *CARNET*s that have *NULL IMAGE*s.

```
SELECT CARNET
FROM IDENTIFICATION
WHERE IMAGE IS
NULL;
```

5.2. Nested

We can *nest* the result of a query to be used in another query. The outside query is called the *outer query*, while inside the query is the *innermost query*. When a nested query return a single attribute and tuple, we can use a $=$ comparator. In any other case, we shall use the **IN** to compare a value within a set of values and returns multiple tuples. In the following example, we can see how we can get all the *COURSE*s *NAME*s and *CREDITS* that have a *GROUP* in the first *SEMESTER* of the *YEAR* 2022.

```
SELECT NAME, CREDITS
FROM COURSE
WHERE ACRONYM IN (
        SELECT ACRONYM
        FROM _GROUP
        WHERE SEMESTER=1 AND YEAR =
2022 );
```

We can also compare multiple values using *tuples* as $(column_1, column_2, \cdots column_n)$ in the outer query's **WHERE** clause. There are other comparison operators that can be used, as shown in Table 5.4.

Operator	**Description**
IN, ANY, SOME	Returns the tuples if we find a value IN the set of values of the inner query.
ALL	With a comparison operator, returns the tuples that are more, less, equal, ... than ALL the innermost tuples.
EXISTS	Returns the tuples if tuples EXIST in the innermost query.
NOT EXIST	Returns the tuples if tuples does NOT EXIST in the innermost query.
NOT EXIST	Returns TRUE if there are no duplicate tuples in the innermost query.

Figura 5.4: Possible compators with nested queries with multiple tuples

- We can use an outer query attribute in an innermost query, we must use an **ALIAS**. The ALIAS can also be used to reduce ambiguities if both tables have the same name.

- Nested queries are *correlated* if the innermost query references an attribute of the outermost query.

- Nested queries that have a SELECT-FROM-WHERE that use = or **IN** can always be written as a single block query.

5.3. JOIN

We can **JOIN** tables (can be more than two) together in the **FROM** clause of a **SELECT** operation. We can see in the following example how we can specify a join, to find the *GROUP*'s information for those given in the first *SEMESTER* of the *YEAR* 2022.

```
SELECT NAME, CREDITS, NUMBER
FROM COURSE AS C JOIN, _GROUP AS G ON C.ACRONYM =
G.ACRONYM
WHERE SEMESTER = 1 AND YEAR = 2022;
```

There are different types of JOINs. We shall assume that the relation R is joined to S, such that R is the leftmost relation and S the rightmost one.

- **INNER JOIN or JOIN**: JOINs relations R and S on a pair of attributes that must have the same values. Each pair of attributes is included only once in the resulting relation.
- **LEFT OUTER JOIN OR LEFT JOIN**: JOINs the relations R and S, ensuring that even if R does not have a matching tuple in S, the tuple in R will appear in the resulting relation with NULL values for S's attributes.
- **RIGHT OUTER JOIN OR RIGHT JOIN**: JOINs the relations R and S, ensuring that even if S does not have a matching tuple in R, the tuple in S will appear in the resulting relation with NULL values for R's attributes.
- **FULL OUTER JOIN OR FULL JOIN**: JOINs the relations R and S, ensuring that even if R or S does not have a matching tuple in the other relation, the tuple will appear in the resulting relation with NULL values.
- **NATURAL <TYPE> JOIN:** A JOIN with no condition specified as the pair of attributes with the same name in R and S are joined once in the resulting relation. The type can be blank (for INNER JOIN), LEFT, RIGHT or FULL. This is similar to an EQUIJOIN in relational algebra.

5.4. Aggregate functions

With *aggregate functions* information from several tuples can be summarized into one. NULL values are discarded in aggregate functions except for **COUNT(*)**.

Function	Description
COUNT	COUNTs the number of tuples in a query.
SUM	SUMs the values of a column in a query (must be numeric).
MAX	Gets the MAX value of a column in a query (must be numeric).
MIN	Gets the MIN value of a column in a query (must be numeric).
AVG	Gets the AVG of the values of a column in a query (must be numeric).

Figura 5.5: Aggregate functions

There are several ways we can use these functions. For example, we can gather the SUM, MAX, MIN and AVG of *CREDITS* for *COURSE*s.

```
SELECT MAX(CREDITS), MIN(CREDITS),
SUM(CREDITS),
AVG(CREDITS) AS AVG_CREDITS
FROM COURSE;
```

We can also count the number of current *COURSE*s saved in the database with the following query.

```
SELECT COUNT(*)
FROM COURSE;
```

5.5. More clauses SELECT

The full select clause has the following clauses (the order of execution is detailed with the elevated number).

```
SELECT[6] <attribute list>
FROM[1] <table list>
WHERE[2] <condition>
GROUP BY[3] <grouping
condition>
HAVING[4] <group condition>
ORDER BY[5] <attribute list>;
```

Thus, the description for the two new clauses (GROUP BY and HAVING) is the following:

- **GROUP BY:** Groups tuples based on a grouping condition. The attributes within the grouping attribute must appear in the SELECT clause, with the desired aggregate functions. A separate GROUP is created for NULL values in a condition.
- **HAVING:** Details a condition to filter groups. It is different from the WHERE clause that filters tuples, while the HAVING clause removes groups.

We can see an example of the **GROUP BY** clauses in the following example. Here, we add an *AREA_NUMBER* to *COURSE*s that has the AREA in which the course is given. For example, computer science courses are within the engineering area. Thus, we group all the *COURSE*s by the area and then gather this number, the number of courses given by area and the average number of *CREDITS*.

```
SELECT AREA_NUMBER, COUNT(*), AVG (CREDITS)
FROM COURSE
GROUP BY AREA_NUMBER;
```

The following example shows how to filter with the **HAVING** clause. This adds from the previous example, where now the *AREA_NUMBER* that have an average less than or equal to 3 are filtered from the results.

```
SELECT AREA_NUMBER, COUNT(*), AVG (CREDITS)
FROM COURSE
GROUP BY AREA_NUMBER
HAVING AVG (CREDITS) > 3;
```

5.6. Assertions

General constraints can be specified using *declarative assertions* using the **CREATE ASSERTION** statement. The following shows an example of an assertion named *POSITIVE_CREDITS* that checks that no *COURSE*s have non-positive *CREDIT*s.

```
CREATE ASSERTION POSITIVE_CREDITS
CHECK ( NOT EXISTS ( SELECT *
                     FROM COURSE
                     WHERE CREDITS < 1));
```

- This constraint is only checked when tuples are inserted or updated in a specific table.
- **CREATE ASSERTION** should only be used when it is not possible to define a **CHECK**.
- The technique to write an ASSERTION is to define a query that selects all the tuples that would violate the condition, by using the **NOT EXISTS** clause.
- The form of an assertion is:

```
CREATE ASSERTION <name>
CHECK (<query that selects all tuples that violates a
condition>);
```

- We cannot define this type of constraints in SQL Server.

5.7. Triggers

Triggers [1] are constraints that are used to *monitor updates, maintain consistency of derived data and update derived data* the database. We can create this constraint with the **CREATE TRIGGER** command. Triggers have three components:

- **EVENTS (E):** The events (e.g., INSERTs, UPDATEs, DELETEs, temporal period time) that activate the trigger. The trigger can be activated **BEFORE**, **INSTEAD OF**, or **AFTER** the operation that activates the trigger is executed.
- **CONDITION (C):** Determines if the action rule shall be executed if a condition is true, detailed in the **WHERE** clause. This component is optional and if it is not defined, the action will be executed once the event is activated.
- **ACTION (A):** SQL statements that are executed when the trigger is activated.

If a trigger has the optional **FOR EACH ROW** clause, then it is activated separately for each tuple and is known as a *row-level trigger*. If not, it will be activated once per statement and be known as a *statement-level trigger*.

[1] `https://docs.microsoft.com/en-us/sql/relational-databases/triggers/dml-triggers`

5.8. Views

A *view* [2] is a table that is defined from other tables (*base tables*), creating a *virtual table* that is not physically stored in the database. Thus there are limitations to updating views, but not querying views. The tables used in the view are called the *defining tables*. The following is an example of a view named *COURSE_GROUPS* that saves the *NAME*, *ACRONYM* and number of groups of *COURSE*s for each *COURSE*.

```
CREATE VIEW COURSE_GROUPS (NAME, ACRONYM, NUM-
BER_GROUPS)
AS SELECT NAME, G.ACRONYM, COUNT(*)
FROM COURSE AS C JOIN _GROUP AS G ON C.ACRONYM =
G.ACRONYM
GROUP BY NAME, G.ACRONYM;
```

We can execute queries over views. We can see an example where all the course names and number of groups are retrieved for the view.

```
SELECT NAME, NUM-
BER_GROUPS
FROM COURSE_GROUPS;
```

- If none of the view attributes are results of aggregate functions, the view name attributes do not need to be specified as they are inherited.
- Views simplify certain queries and serve as security or authentication mechanisms. With the latter, only certain access can be given to attributes.
- As views have to be up-to-date, they are materialized when a query is specified to the view.
- We can remove views with the **DROP VIEW** command.
- Usually, views data can be modified if it only has only one table. This is the case for SQL Server [3].
- Within SSMS, we can see all the views in *Databases* > *Views*, as shown in Fig. 5.6.

5.9. Stored procedures

With the previous commands, there is an assumption that queries run in the application server. However, we can execute the program in the database server using stored procedures [4] or functions [5] (*persistent stored modules*).

In the following example, we can see how a stored procedure is defined that receives a *COURSE*s' *ACRONYM* and gives the *NAME*.

We can call persistent stored modules using the **EXECUTE** or **EXEC** command, as follows.

[2] https://docs.microsoft.com/en-us/sql/relational-databases/views/views

[3] https://docs.microsoft.com/en-us/sql/relational-databases/views/modify-data-through-a-view

[4] https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine

[5] https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions

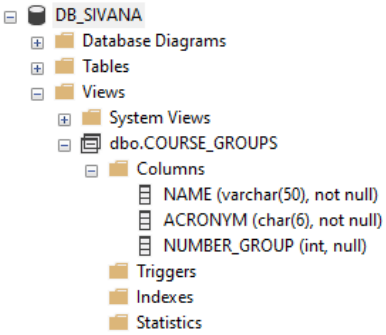


Figura 5.6: Views in SSMS

```
CREATE PROCEDURE GetNameCourse (@ACRONYM
CHAR(6))
AS BEGIN
        SELECT NAME
        FROM COURSE
        WHERE ACRONYM = @ACRONYM
END;
```

```
EXECUTE GetNameCourse @ACRONYM =
'CI0127'
```

- Parameters have a name, a type (SQL data types) and a mode. The mode is IN (input only), OUT (output only), INOUT (both input and output). Thus, if a parameter has IN it can receive the parameter, while OUT indicates that it returns it.
- The difference with both of these modules is that functions return data, while stored procedures do not.
- These persistent stored modules are stored persistently in the database server and are useful if a database program is needed by several applications (reduces effort and improved modularity), in certain situations reduce data transfer costs between client and server, and enhance modeling power.
- Within the declaration of the persistent stored modules, we can have control-flow statements [6].
- Within SSMS, we can see all the stored procedures and functions in *Databases* > *Programmability* > *Stored Procedures* and *Databases* > *Programmability* > *Functions*, respectively. This is shown as shown in Fig. 5.7.

[6] https://docs.microsoft.com/en-us/sql/t-sql/language-elements/control-of-flow

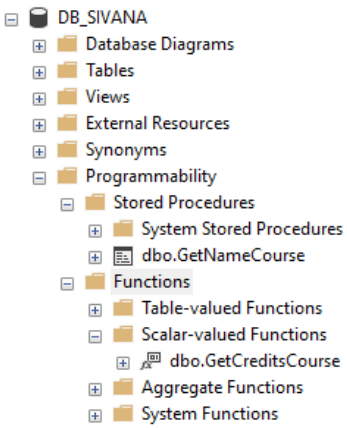


Figura 5.7: Persistent stored modules in SSMS

Parte IV

Cálidad

Capítulo 6

Evaluación de la calidad del diseño

6.1. Design guidelines

While designing a database, we have yet to formally define the goodness or appropriateness of the design. Thus, we can use both guidelines and measure the quality of the design. There are two levels we can define goodness:

1. The *logical or conceptual level* on how users interpret the schemas.

2. The *implementation or physical level* on how tuples are stored and updated.

The goals are:

- **Information preservation:** Preserves the information (e.g., attributes, entities, relationships) described in the models from the conceptual design to the logical design mapping.

- **Minimum redundancy:** Minimizes saving the same information to reduce storage, inconsistencies and the need of multiple updates. Though, sometimes some form of redundancy may be desired depending on the application (NoSQL).

Following, we shall define good general informal guidelines for design.

Guideline #1: Clear semantics

We can informally measure how *semantically* easy to understand are the database elements (entities and relationship) in the design. Thus, every element should have a straightforward meaning that represents only one concept.

For example, the following table combines multiple concepts of students, schools and faculty into only one relation named *STUDENT_SCHOOL_FACULTY*. Therefore, it is hard to understand which attributes are related to which concept. For example, is the student's name, the school's name or the faculty's name? Is the number of students for the school or the faculty? Which acronym are they referring to?

STUDENT_SCHOOL_FACULTY

Email	Name	Phone_number	Acronym	Number_students

Guideline #2: Avoid redundancy

If data in the database is redundant (storing the result of natural joins), we are wasting space and may lead to several anomalies. For example, the following table shows how the name, phone and faculty name for the "ECCI"" school is duplicated in t_1 and t_2. Furthermore, the name of "Bob Benavidez" is repeated, though they can be recognized only with their email.

STUDENT_SCHOOL_FACULTY

Student_mail	Student_name	School_acronym	School_phone	Faculty_name
alicia.armando23	Alicia Armando	ECCI	2511-8000	Ingenería
bob.benavidez	Bob Benavidez	ECCI	2511-8000	Ingenería
bob.benavidez	Bob Benavidez	EMat	2511-6551	Ciencias Básicas
carlos.calvo	Carlos Calvo	EAN	2511-9180	Ciencias Sociales
daniela.delgado	Daniela Delgado	EMat	2511-6551	Ciencias Básicas

There are different types of anomalies that can occur:

- **Insertion:** While inserting tuples, inconsistencies can occur with new data that is added. For example, when we add another student of the "ECCI"" school, we must ensure that the phone number and faculty name are correct in all the tuples. Furthermore, if we want to add information only about the school without students it is not possible due to key constraints.
- **Deletion:** While deleting certain tuples, we can lose all the information in the database. For example if we delete "Carlos Calvo", we lose all the information related to the school "EAN".
- **Modification:** While modifying tuples, we have to always modify all redundant references. For example, if we modify the phone number of the "ECCI"" school we must update the phone number for tuple t_1 and t_2 else data will be inconsistent.

Guideline #3: Reduce NULLs

With many tuples not applying in a relation, space can be wasted and the differences between NULL types may not be understandable (does not apply, unknown or absent). For example, in the following table saving in an attribute the position of a student in the association will probably lead to many NULLs as there are less than ten members per year. Furthermore, Fax information probably does not exist for most students. Their grade in kindergarten is probably not known for most students. Finally, most students will also probably not link their instagram account within the institution.

STUDENT

Email	Name	Position_association	Fax_number	Kindergarden_grade	Instagram

Guideline #4: Consider false tuples

We must ensure when tables are created, the result of joining them gives a correct result. Thus, we must use attributes that are appropriately related such as keys and foreign keys. If not, it generates false tuples that do not represent valid data (a.k.a sussy). For example, if we have the following tables *STUDENT* and *SCHOOL*. The "correct" tuples are shown for guideline #2.

STUDENT

Email	Name	Faculty_name
alicia.armando23	Alicia Armando	Ingenería
bob.benavidez	Bob Benavidez	Ingenería
bob.benavidez	Bob Benavidez	Ciencias Básicas
carlos.calvo	Carlos Calvo	Ciencias Sociales
daniela.delgado	Daniela Delgado	Ciencias Básicas

SCHOOL

School_acronym	School_phone	Faculty_name
ECCI	2511-8000	Ingenería
EMat	2511-6551	Ciencias Básicas
EAN	2511-9180	Ciencias Sociales
ECCC	2511-3600	Ciencias Sociales

If we join both tables on the attribute *Faculty_name*, we will generate a false tuple marked in red that did not previously exist.

STUDENT_SCHOOL_FACULTY

Student_mail	Student_name	School_acronym	School_phone	Faculty_name
alicia.armando23	Alicia Armando	ECCI	2511-8000	Ingenería
bob.benavidez	Bob Benavidez	ECCI	2511-8000	Ingenería
bob.benavidez	Bob Benavidez	EMat	2511-6551	Ciencias Básicas
carlos.calvo	Carlos Calvo	EAN	2511-9180	Ciencias Sociales
carlos.calvo	Carlos Calvo	ECCC	2511-3600	Ciencias Sociales
daniela.delgado	Daniela Delgado	EMat	2511-6551	Ciencias Básicas

All guidelines

Guideline #1: While designing the relational schema, the attributes should be easy to understand and explain. Thus, every entity or relationship has a straightforward meaning representing only one concept.

Guideline #2: Design to avoid insertion, deletion or modification anomalies. If not possible, clearly state the anomalies and update the database correctly.

Guideline #3: Avoid NULL attributes while possible. If not possible, ensure that they do not apply to the majority of tuples in the relation.

Guideline #4: Design relation schemas so that they can be joined correctly, using the appropriate primary and foreign keys that do not generate false tuples.

6.2. Functional dependencies

Based on these informal rules, we can use functional dependencies to formally define some of these issues.

A *functional dependency* is a constraint denoted by $X \rightarrow Y$ for a set of attributes X and Y for a relation R.It can be defined when any two tuples that have $t_1[X] = t_2[X]$ must also have $t_1[Y] = t_2[Y]$.

- The attributes of Y are determined (i.e., functionally dependent) on the set of attributes X.
- It can be abbreviated into FD or f.d.

- $X \to Y$ does not indicate that $Y \to X$ is true.
- All candidate keys X can determine the attributes of any attribute Y in R. Therefore, $X \to R$.
- This constraint is for any possible relation state r of R. Thus, it is a property of the relation R and not for a state.
- We *cannot* infer automatically a FD based on the state of a relation, but we may indicate that a DF *may* exist. Any counter example in a state (old, current or old) can disprove the DF.

6.3. Normal forms

A schema can be normalized based on a series of tests that specify criteria for a *normal form*.

- Combining normal forms, ensuring to not create false tuples (*nonadditive join or lossless join*) and preserving most dependencies (*dependency preservation property*), we can improve the database design. We do not need to ensure the dependency preservation property in the resulting design.
- The process in which we pass a non-compliant design to a normal form is called *normalization*. The inverse process, in which a design normal form is lowered is called *denormalization*.
- There are various normal forms, but we shall see in detail the first three: 1NF, 2NF and 3NF.
- Though some normal forms do not need to satisfy the requirements of a lower level normal form, for historical reasons it is customary that the sequence is followed. Thus, 3NF already satisfies 1NF.

1NF

Every attribute within a relation must be *atomic* (a single value for any attribute) or nested relations (tuples with relations within it). This is considered a requirement for any flat relational model. Every attribute that does not follow this restriction is moved to another relation that has a foreign key to the primary key of this relation. Thus separating the non-1NF relation into two 1NF relations.

2NF

Every attribute that is part of a candidate key is a *prime attribute*. While, attributes are non prime if they are not prime. 2NF expects every non prime attribute in a relation to only be fully dependent on the primary key of the relation. Thus, for every primary key a new relation is created with all the nonprime attributes that depend on that primary key (based on the FD).

3NF

There is a transitive dependency from $X \to Z$, when there exists a FD from $X \to Y$ and $Y \to Z$ where Y is a non prime attribute.

3nd expects that every non prime attribute in a relation has a FD with the primary key. As in, there can be no transitive dependencies from the primary key of the relation. To normalize this dependency, we create a new relation with the primary key as the Y in the transitive relation and leave the attribute in the previous relation.

6.4. Review

Guías de diseño informales: Errores

1. Se tiene el siguente diseño relacional de un sistema de bugs.

BUGS_CODE:

BugID	BugDescription	LineNumber	FileName

   (a) ¿Cuál de los siguientes guías de diseño informales no se cumple?
   ◯ Semántica clara ◯ Reducir redundancia ◯ Reducir NULLs ◯ Considerar tuplas falsas ◯ Ninguno

   (b) Justifique porque se cumplen o no esas guías de diseño informales.

   (c) Proponga las mejoras de diseño correspondientes en el caso de que no sigan las guías de diseño informal.

2. Se tiene el siguente diseño relacional de un sistema de vacunación.

VACCINE_HOSPITAL_PACIENT:

ID	Name	PhoneNumber	Dosage	Notes

   (a) ¿Cuál de los siguientes guías de diseño informales no se cumple?
   ◯ Semántica clara ◯ Reducir redundancia ◯ Reducir NULLs ◯ Considerar tuplas falsas ◯ Ninguno

   (b) Justifique porque se cumplen o no esas guías de diseño informales.

   (c) Proponga las mejoras de diseño correspondientes en el caso de que no sigan las guías de diseño informal.

3. Se tiene el siguente diseño relacional de un sistema que guarda resultados de partidas de video juegos.

PLAYER:

ID	Name

MATCH_PLAYER:

MatchID	PlayerID	Kills	Deaths	Assists

   (a) ¿Cuál de los siguientes guías de diseño informales no se cumple?
   ◯ Semántica clara ◯ Reducir redundancia ◯ Reducir NULLs ◯ Considerar tuplas falsas ◯ Ninguno

(b) Justifique porque se cumplen o no esas guías de diseño informales.

(c) Proponga las mejoras de diseño correspondientes en el caso de que no sigan las guías de diseño informal.

4. Se tiene el siguente diseño relacional de un sistema de un banco.

BANK:

ID	PhoneNumber	Location

BANK_CLIENT:

BankID	ClientID	ResidenceTown

(a) ¿Cuál de los siguientes guías de diseño informales no se cumple?
◯ Semántica clara ◯ Reducir redundancia ◯ Reducir NULLs ◯ Considerar tuplas falsas ◯ Ninguno

(b) Justifique porque se cumplen o no esas guías de diseño informales.

(c) Proponga las mejoras de diseño correspondientes en el caso de que no sigan las guías de diseño informal.

5. Se tiene el siguente diseño relacional de un sistema de votación electrónico.

CITIZEN:

ID	ResidenceTown	BirthDate

VOTE:

VoteID	ResidenceTown	ElectedCandidate

(a) ¿Cuál de los siguientes guías de diseño informales no se cumple?
◯ Semántica clara ◯ Reducir redundancia ◯ Reducir NULLs ◯ Considerar tuplas falsas ◯ Ninguno

(b) Justifique porque se cumplen o no esas guías de diseño informales.

(c) Proponga las mejoras de diseño correspondientes en el caso de que no sigan las guías de diseño informal.

Dependencias funcionales: Estados

1. Se tiene los siguientes estados dentro de una relación:

A	**B**	**C**
10	x	hello
20	y	bonjour
30	y	hola

(a) ¿Existe una dependencia funcional de $A \to B$? ◯ Si ◯ Puede ◯ No

(b) ¿Existe una dependencia funcional de $B \to A$? ◯ Si ◯ Puede ◯ No

(c) ¿Existe una dependencia funcional de $A \to C$? ◯ Si ◯ Puede ◯ No

(d) ¿Existe una dependencia funcional de $C \to A$? ◯ Si ◯ Puede ◯ No

(e) ¿Existe una dependencia funcional de $B \to C$? ◯ Si ◯ Puede ◯ No

A	B	C	D
triangle	100	red	3
circle	20	white	0
rectangle	100	blue	4
circle	30	white	0
square	90	black	4

(f) ¿Existe una dependencia funcional de $\{A, B\} \rightarrow C$? ◯ Si ◯ Puede ◯ No

2. Se tiene los siguientes estados dentro de una relación:

(a) ¿Existe una dependencia funcional de $A \rightarrow B$? ◯ Si ◯ Puede ◯ No

(b) ¿Existe una dependencia funcional de $B \rightarrow A$? ◯ Si ◯ Puede ◯ No

(c) ¿Existe una dependencia funcional de $A \rightarrow C$? ◯ Si ◯ Puede ◯ No

(d) ¿Existe una dependencia funcional de $C \rightarrow A$? ◯ Si ◯ Puede ◯ No

(e) ¿Existe una dependencia funcional de $B \rightarrow C$? ◯ Si ◯ Puede ◯ No

(f) ¿Existe una dependencia funcional de $C \rightarrow B$? ◯ Si ◯ Puede ◯ No

(g) ¿Existe una dependencia funcional de $A \rightarrow D$? ◯ Si ◯ Puede ◯ No

(h) ¿Existe una dependencia funcional de $D \rightarrow A$? ◯ Si ◯ Puede ◯ No

(i) ¿Existe una dependencia funcional de $\{A, B\} \rightarrow C$? ◯ Si ◯ Puede ◯ No

(j) ¿Existe una dependencia funcional de $\{A, B\} \rightarrow D$? ◯ Si ◯ Puede ◯ No

(k) ¿Existe una dependencia funcional de $\{C, B\} \rightarrow D$? ◯ Si ◯ Puede ◯ No

(l) ¿Existe una dependencia funcional de $\{B, C, D\} \rightarrow A$? ◯ Si ◯ Puede ◯ No

3. Se tiene los siguientes estados dentro de una relación:

A	B	C	D	E
Costa Rica	America	español	True	S
Japón	Asia	japonés	False	L
España	Europa	español	True	M
Kenya	Africa	swahili	False	M
Australia	Oceania	inglés	False	S

(a) ¿Existe una dependencia funcional de $A \rightarrow B$? ◯ Si ◯ Puede ◯ No

(b) ¿Existe una dependencia funcional de $B \rightarrow A$? ◯ Si ◯ Puede ◯ No

(c) ¿Existe una dependencia funcional de $A \rightarrow C$? ◯ Si ◯ Puede ◯ No

(d) ¿Existe una dependencia funcional de $C \rightarrow A$? ◯ Si ◯ Puede ◯ No

(e) ¿Existe una dependencia funcional de $C \rightarrow D$? ◯ Si ◯ Puede ◯ No

(f) ¿Existe una dependencia funcional de $D \rightarrow B$? ◯ Si ◯ Puede ◯ No

(g) ¿Existe una dependencia funcional de $E \rightarrow A$? ◯ Si ◯ Puede ◯ No

(h) ¿Existe una dependencia funcional de $E \rightarrow B$? ◯ Si ◯ Puede ◯ No

(i) ¿Existe una dependencia funcional de $\{B, C\} \rightarrow A$? ◯ Si ◯ Puede ◯ No

(j) ¿Existe una dependencia funcional de $\{C, D\} \rightarrow B$? ◯ Si ◯ Puede ◯ No

(k) ¿Existe una dependencia funcional de $\{D, E\} \rightarrow B$? ◯ Si ◯ Puede ◯ No

(l) ¿Existe una dependencia funcional de $\{C, D, E\} \rightarrow B$? ◯ Si ◯ Puede ◯ No

(m) ¿Existe una dependencia funcional de $\{B, C, D\} \rightarrow E$? ◯ Si ◯ Puede ◯ No

(n) ¿Existe una dependencia funcional de $\{B, C, D\} \rightarrow A$? ◯ Si ◯ Puede ◯ No

(ñ) ¿Existe una dependencia funcional de $\{A, B, C, D\} \rightarrow E$? ◯ Si ◯ Puede ◯ No

(o) ¿Existe una dependencia funcional de $\{A, B, D, E\} \rightarrow C$? ◯ Si ◯ Puede ◯ No

Dependencias funcionales: Clausuras

1. Para la relación $\alpha(A, B, C)$ se tienen las siguientes dependencias funcionales (FD):

$$A \rightarrow B \quad \text{(FD1)}$$
$$\{A, B\} \rightarrow C \quad \text{(FD2)}$$
$$C \rightarrow B \quad \text{(FD3)}$$

   (a) Detalle la clausura de $\{A\}$.

   (b) ¿$\{A\}$ es una posible superllave de α? ◯ Si ◯ No

   (c) Detalle la clausura de $\{C\}$.

   (d) ¿$\{C\}$ es una posible superllave de α? ◯ Si ◯ No

2. Para la relación $\beta(A, B, C, D, E)$ se tienen las siguientes dependencias funcionales (FD):

$$\{A, D\} \rightarrow B \quad \text{(FD1)}$$
$$\{C, D\} \rightarrow E \quad \text{(FD2)}$$
$$B \rightarrow A \quad \text{(FD3)}$$
$$A \rightarrow D \quad \text{(FD4)}$$
$$\{D, B\} \rightarrow C \quad \text{(FD5)}$$

   (a) Detalle la clausura de $\{B, D\}$.

   (b) ¿$\{B, D\}$ es una posible superllave de β? ◯ Si ◯ No

   (c) Detalle la clausura de $\{C, D\}$.

   (d) ¿$\{C, D\}$ es una posible superllave de β? ◯ Si ◯ No

   (e) Detalle la clausura de $\{A\}$.

   (f) ¿$\{A\}$ es una posible superllave de β? ◯ Si ◯ No

3. Para la relación $\gamma(R, S, T, U, V, W, X, Y, Z)$ se tienen las siguientes dependencias funcionales (FD):

$$\{T, U, V\} \rightarrow W, X \quad \text{(FD1)}$$
$$W \rightarrow S \quad \text{(FD2)}$$
$$R \rightarrow S, V \quad \text{(FD3)}$$

$$\{X, Z\} \to Y \quad \text{(FD4)}$$

$$Z \to T, R \quad \text{(FD5)}$$

$$\{S, R\} \to U \quad \text{(FD6)}$$

(a) Detalle la clausura de $\{T, U, V\}$.

(b) ¿$\{T, U, V\}$ es una posible superllave de γ? ◯ Si ◯ No

(c) Detalle la clausura de $\{R, T, Y\}$.

(d) ¿$\{R, T, Y\}$ es una posible superllave de γ? ◯ Si ◯ No

(e) Detalle la clausura de $\{Z\}$.

(f) ¿$\{Z\}$ es una posible superllave de γ? ◯ Si ◯ No

(g) Detalle la clausura de $\{R, W\}$.

(h) ¿$\{R, W\}$ es una posible superllave de γ? ◯ Si ◯ No

Dependencias funcionales: Llaves

Para cada una de las siguientes preguntas, marque todas las opciones posibles.

1. Para la relación $\delta(X, W, Y, Z)$ se tienen las siguientes dependencias funcionales (FD):

$$Y \to Z \quad \text{(FD1)}$$

$$\{X, W\} \to Z \quad \text{(FD2)}$$

$$Y \to X \quad \text{(FD3)}$$

(a) Marque cuales de las siguientes son superllaves de δ. ◯ X ◯ W ◯ Y ◯ Z ◯ $\{X, W\}$ ◯ $\{X, Y\}$ ◯ $\{X, Y, Z\}$

(b) Marque cuales de las siguientes son llaves canidatas de δ. ◯ X ◯ W ◯ Y ◯ Z ◯ $\{X, W\}$ ◯ $\{X, Y\}$ ◯ $\{X, Y, Z\}$

2. Para la relación $\epsilon(F, G, H, I, J, K)$ se tienen las siguientes dependencias funcionales (FD):

$$\{F, G\} \to K \quad \text{(FD1)}$$

$$K \to J \quad \text{(FD2)}$$

$$\{H, J\} \to F \quad \text{(FD3)}$$

$$F \to H, I \quad \text{(FD4)}$$

$$I \to G, F \quad \text{(FD4)}$$

(a) Marque cuales de las siguientes son superllaves de ϵ. ◯ F ◯ G ◯ H ◯ I ◯ J ◯ K ◯ $\{F, G\}$ ◯ $\{G, K\}$ ◯ $\{H, J\}$ ◯ $\{J, K\}$ ◯ $\{G, H, J\}$ ◯ $\{G, J, K\}$

(b) Marque cuales de las siguientes son llaves canidatas de ϵ. ◯ F ◯ G ◯ H ◯ I ◯ J ◯ K ◯ $\{F, G\}$ ◯ $\{G, K\}$ ◯ $\{H, J\}$ ◯ $\{J, K\}$ ◯ $\{G, H, J\}$ ◯ $\{G, J, K\}$

3. Para la relación $\zeta(A, B, C, D, E, F, G, H, I, J)$ se tienen las siguientes dependencias funcionales (FD):

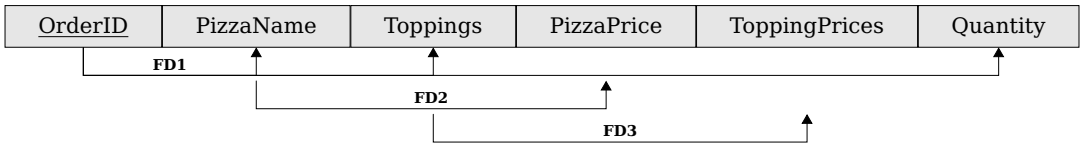
$$\{C, E\} \rightarrow I \quad \text{(FD1)}$$
$$A \rightarrow J \quad \text{(FD2)}$$
$$\{A, D\} \rightarrow H \quad \text{(FD3)}$$
$$\{H, I, J\} \rightarrow F \quad \text{(FD4)}$$
$$D \rightarrow E, G \quad \text{(FD5)}$$
$$\{D, J\} \rightarrow B \quad \text{(FD6)}$$

(a) Marque cuales de las siguientes son superllaves de ζ. ◯ A ◯ E ◯ I
◯ $\{A, B\}$ ◯ $\{C, E\}$ ◯ $\{C, D\}$ ◯ $\{D, E\}$ ◯ $\{E, G\}$ ◯ $\{F, J\}$ ◯ $\{I, G\}$
◯ $\{A, B, C\}$ ◯ $\{A, C, D\}$ ◯ $\{A, C, E\}$ ◯ $\{D, E, G\}$ ◯ $\{H, I, J\}$
◯ $\{A, B, C, D\}$ ◯ $\{A, D, G, J\}$ ◯ $\{G, H, I, J\}$ ◯ $\{A, D, E, F, G\}$ ◯ $\{E, F, G, H, I\}$

(b) Marque cuales de las siguientes son llaves canidatas de ζ. ◯ A ◯ E ◯ I
◯ $\{A, B\}$ ◯ $\{C, E\}$ ◯ $\{C, D\}$ ◯ $\{D, E\}$ ◯ $\{E, G\}$ ◯ $\{F, J\}$ ◯ $\{I, G\}$
◯ $\{A, B, C\}$ ◯ $\{A, C, D\}$ ◯ $\{A, C, E\}$ ◯ $\{D, E, G\}$ ◯ $\{H, I, J\}$
◯ $\{A, B, C, D\}$ ◯ $\{A, D, G, J\}$ ◯ $\{G, H, I, J\}$ ◯ $\{A, D, E, F, G\}$ ◯ $\{E, F, G, H, I\}$

Formas normales

1. Se tienen las siguientes relaciones con su conjunto de dependencias funcionales.



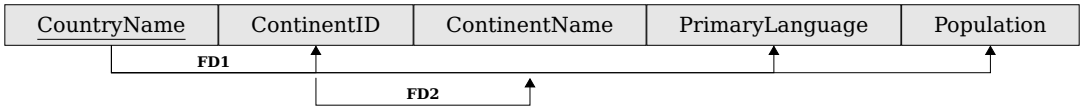
La siguiente tabla muestra datos de ejemplo que puede guardar la relación.

OrderID	PizzaName	Toppings	PizzaPrice	ToppingPrices	Quan
29301	Margherita	{Black olives, mushrooms}	$10	{2,1.5}	3
435534	Cheese	{Hot Sauce, Squids, Shrimps}	$7	{1,4, $3}	10
343243	Chocolate	{Marshmellows}	$15	{$1}	1
233123	Cheese	{Pineapple}	$7	{$2}	5

(a) ¿La relación está en $1NF$? ◯ Si ◯ No
(b) Justifique formalmente porque está o no está en $1NF$.
(c) ¿La relación está en $2NF$? ◯ Si ◯ No
(d) Justifique formalmente porque está o no está en $2NF$.
(e) ¿La relación está en $3NF$? ◯ Si ◯ No
(f) Justifique formalmente porque está o no está en $3NF$.
(g) Normalice hasta la tercera forma normal, detallando cada paso.

2. Se tienen las siguientes relaciones con su conjunto de dependencias funcionales.

La siguiente tabla muestra datos de ejemplo que puede guardar la relación.



CountryName	ContinentID	ContinentName	PrimaryLanguage	Population
Costa Rica	7	America	spanish	5000000
France	3	Europe	french	67000000
Haiti	7	America	french	11000000

(a) ¿La relación está en $1NF$? ◯ Si ◯ No

(b) Justifique formalmente porque está o no está en $1NF$.

(c) ¿La relación está en $2NF$? ◯ Si ◯ No

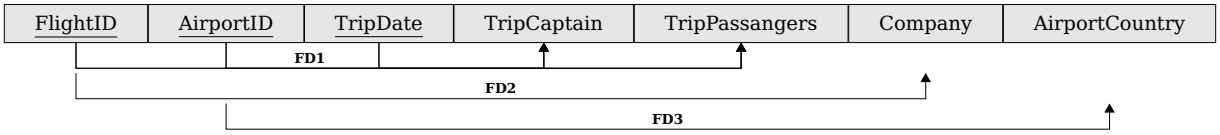
(d) Justifique formalmente porque está o no está en $2NF$.

(e) ¿La relación está en $3NF$? ◯ Si ◯ No

(f) Justifique formalmente porque está o no está en $3NF$.

(g) Normalice hasta la tercera forma normal, detallando cada paso.

3. Se tienen las siguientes relaciones con su conjunto de dependencias funcionales.



La siguiente tabla muestra datos de ejemplo que puede guardar la relación.

FlightID	AirportID	TripDate	TripCaptain	TripPass	Company	AirportCoun
AA 2106	SJO	23/06/2022	J. Doe	120	AMERICAN	Cota Rica
IB 6314	SJO	23/06/2022	A. White	45	IBERIA	Cota Rica
WN3920	FLL	22/06/2022	L. Kim	82	SOUTHWEST	USA

(a) ¿La relación está en $1NF$? ◯ Si ◯ No

(b) Justifique formalmente porque está o no está en $1NF$.

(c) ¿La relación está en $2NF$? ◯ Si ◯ No

(d) Justifique formalmente porque está o no está en $2NF$.

(e) ¿La relación está en $3NF$? ◯ Si ◯ No

(f) Justifique formalmente porque está o no está en $3NF$.

(g) Normalice hasta la tercera forma normal, detallando cada paso.

4. Se tienen las siguientes relaciones con su conjunto de dependencias funcionales.

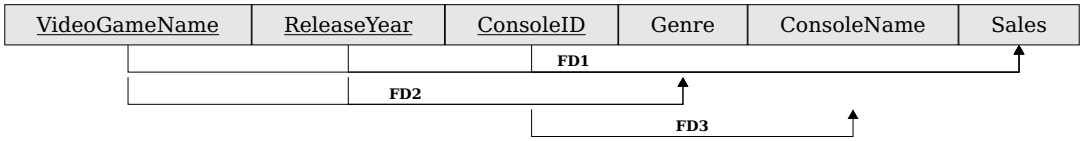
La siguiente tabla muestra datos de ejemplo que puede guardar la relación.

(a) ¿La relación está en $1NF$? ◯ Si ◯ No

(b) Justifique formalmente porque está o no está en $1NF$.

(c) ¿La relación está en $2NF$? ◯ Si ◯ No

(d) Justifique formalmente porque está o no está en $2NF$.



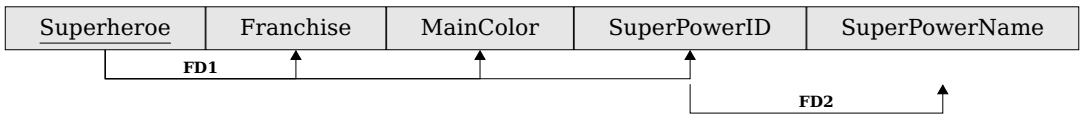
VideoGameName	ReleaseYear	ConsoleID	Genre	ConsoleName	Sales
Breath of the Wild	2017	3002	{Open world, RPG}	Switch	26M
Skyrim	2012	3002	{Open world, RPG}	Switch	2M
Overcooked	2016	2004	{Couch co-op}	PS4	0.5M
Skyrim	2012	1	{Open world, RPG}	PC	9M

(e) ¿La relación está en $3NF$? ◯ Si ◯ No

(f) Justifique formalmente porque está o no está en $3NF$.

(g) Normalice hasta la tercera forma normal, detallando cada paso.

5. Se tienen las siguientes relaciones con su conjunto de dependencias funcionales.



La siguiente tabla muestra datos de ejemplo que puede guardar la relación.

Superheroe	Franchise	MainColor	SuperPowerID	SuperPowerName
Batman	DC	{Black}	1	Money
Superman	DC	{Red,Blue}	2	All
Wonderwoman	DC	{Red,Blue,Gold}	2	All
Iron man	Marvel	{Red,Gold}	3	Mecha
Hulk	Marvel	{Green,Purple}	4	Strength

(a) ¿La relación está en $1NF$? ◯ Si ◯ No

(b) Justifique formalmente porque está o no está en $1NF$.

(c) ¿La relación está en $2NF$? ◯ Si ◯ No

(d) Justifique formalmente porque está o no está en $2NF$.

(e) ¿La relación está en $3NF$? ◯ Si ◯ No

(f) Justifique formalmente porque está o no está en $3NF$.

(g) Normalice hasta la tercera forma normal, detallando cada paso.

Parte V

Funcionamiento físico

Capítulo 7

Organización física de archivos e índices

7.1. Physical storage

Computers can store data in different mediums:

- **Primary storage:** Data stored in main memory, thus the data is stored only while the computer is on (*volatile storage*). This includes cache and RAM. This memory is faster, but is more expensive and has less storage capacity than secondary storage.
- **Secondary storage:** Data stored persistently in the database (*nonvolatile storage*). This includes flash memory (SSD) and magnetic disks (HDD). This storage is slower than primary storage, however, it is cheaper and has more storage capacity.
- **Tertiary storage:** Data stored in removable media that is offline from the main system. This includes DVDs, flash drives, and magnetic tapes.

Databases store *data persistently* as there are large amounts that must be stored for large amounts of time. Thus, they are usually stored in nonvolatile storage with magnetic disks as the main medium of storage for data files. Thus, DBMS tends to have a disk-oriented structure using the primary memory as storage. While the data is being used, it has to be loaded into volatile storage. Databases are also frequently backed up with magnetic tapes.

7.2. Physical database design

DBMS manages the structure in which the data is organized with the *physical database design*. All the information about the database is saved in disk blocks through the operative system (OS). But, the DBMS is in charge of managing the blocks to have more control over efficient access and support recovery.

The general physical database design of a *disk-oriented* is shown in Fig. 7.1. When a query is executed by the *execution engine* that asks for data that is stored in the database. This data is stored into files that are organized into pages. Therefore, the *storage manager* finds the files, pages and records related to the query from memory and disk. To get the data, first the relevant page directory is loaded from disk to memory. Based on the information of the page directory, the relevant page is found and loaded. Finally, the records are found within the page using the header and are given to the query as a result.

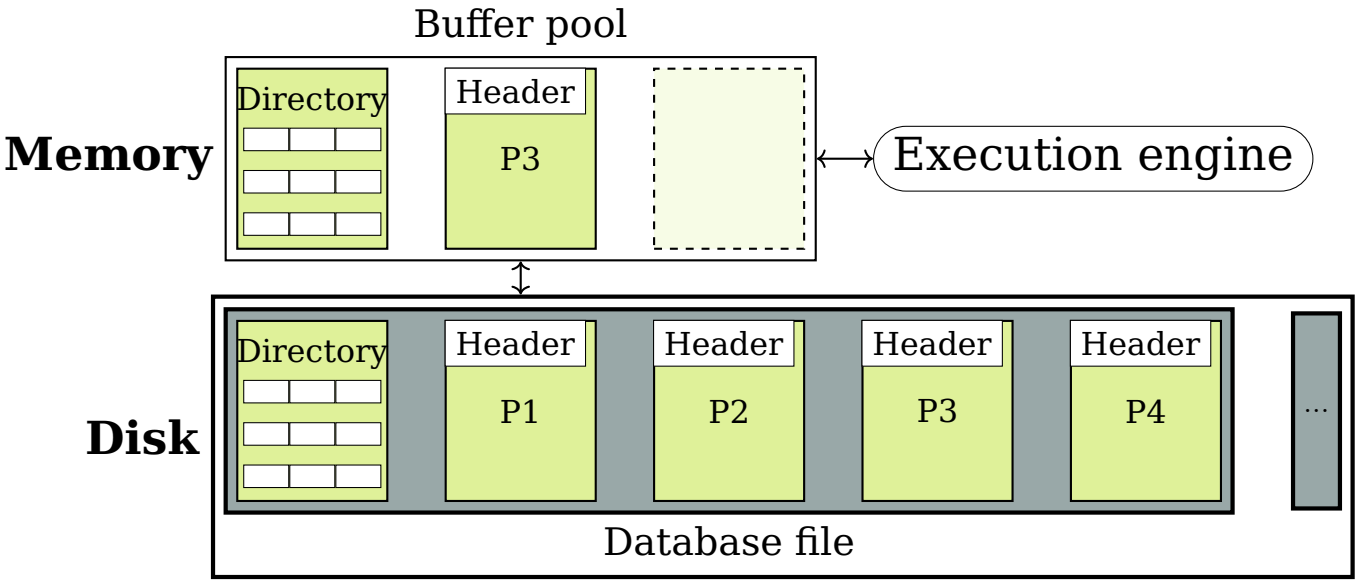


Figura 7.1: Disk-oriented physical database design

In the following sections, the building blocks of the physical memory are detailed: files, pages and records. Their relationship is shown in Fig. 7.2.

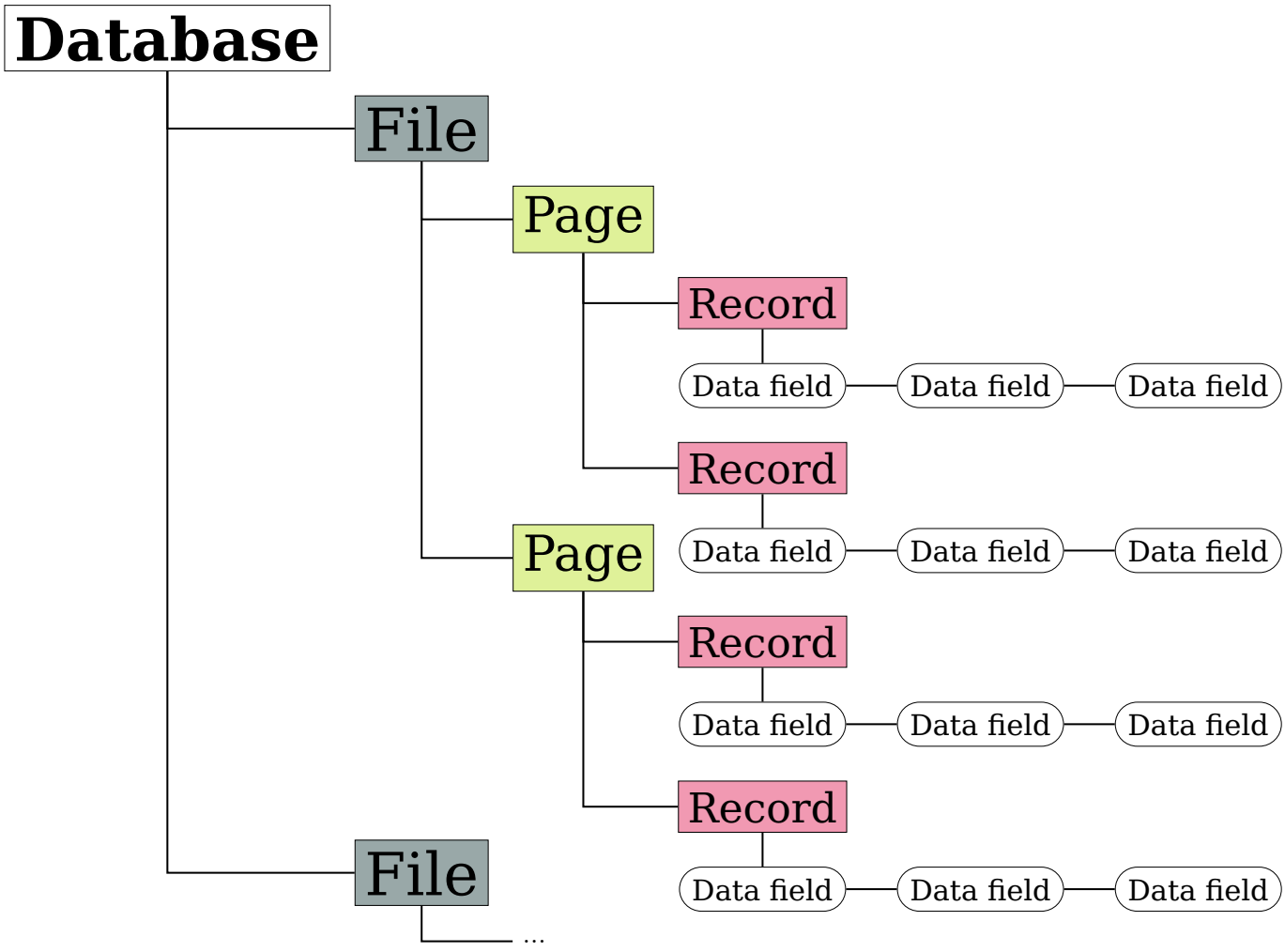


Figura 7.2: Relationship between databases, files, pages and records

7.3. Files

Databases store data in files. These files are saved on disk using the OS, however, only the DBMS knows how to decipher these files. Some DBMS store files in a hierarchy, while others store everything in one file.

7.4. Pages

Each file is logically partitioned into fixed-length storage units used for data allocation and transfer called *pages*. A page is sometimes called a *block*. Pages may contain several records. Every page has a unique identifier.

There are several ways a DBMS can locate a page within a file including database heaps, sorted files, and hashes. DBMS commonly save unordered pages that are tracked using a *page directory* (Fig. 7.3). This directory saves the position for every page, retrieving the specific pages with the identifier. This page directory is the first page within the file, thus it is easily found.

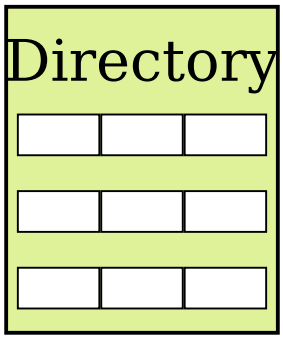


Figura 7.3: Page directory

To store a variable length of records on a page, we can use a *slotted-page structure* or log-based structure. DBMS commonly uses the slotted-based structure, hence we shall further describe it. The slotted-page structure is shown in Fig. 7.4. There is a header at the begging of each page containing the number of records within the page, offset to the last used record, and an array that details the size and location of each record (slot array). Records are stored beside each other in the block, starting from the end. Data is inserted at the end of the free space. Deleted records may be reused or ignored, depending on the DBMS.

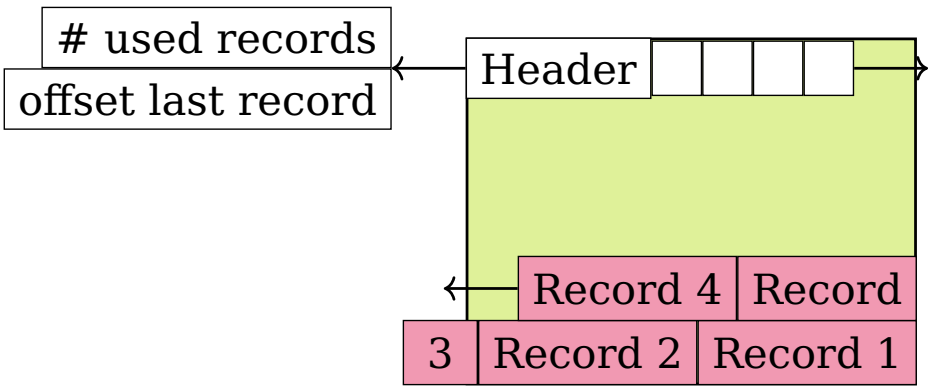


Figura 7.4: Slotted-page structure for a page

7.5. Records

Records are a collection of data items for different fields. They are what we know as tuples or rows in the relational model and SQL, respectively. Each record is uniquely identifiable with an identifier (primary key of the table).

Usually, in a database records of different relations have different sizes. Thus, there are several approaches used to manage these differences:

- **Fixed length.** We assign a number of bytes n for each record.
- **Variable length.** To represent every record with variable length (Fig. 7.5). Each record will have a fixed-length header at the begining of the record with details the offeset (os_x) to the data item and the size of the data item (l_x). Therefore, every data item has the tuple $(offset_x, length_x)$ Followed by this header, the information of the data items are stored (df_x).

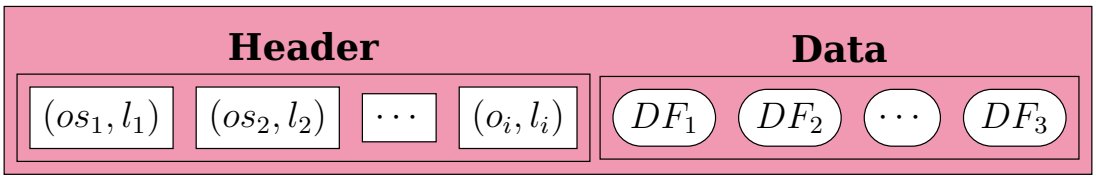
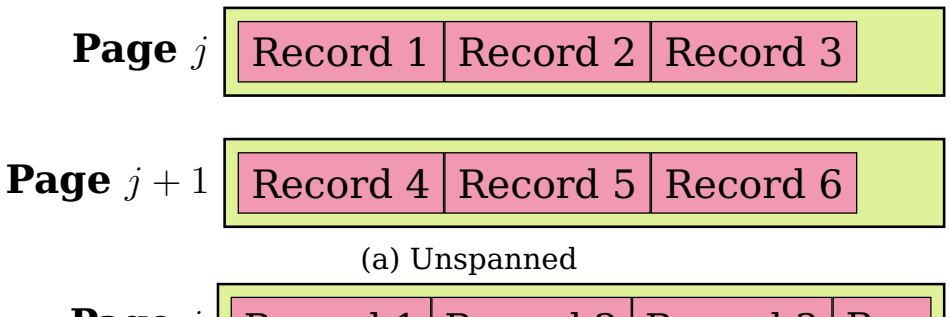


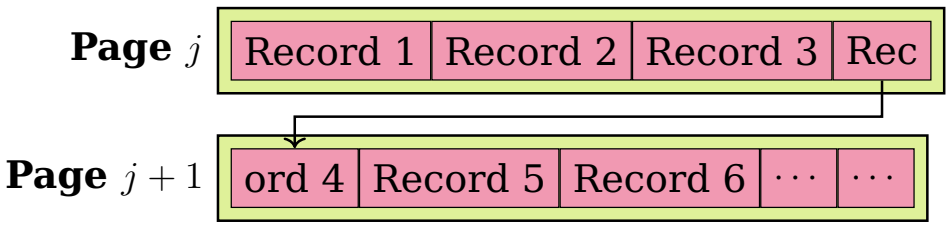
Figura 7.5: Variable length record structure

As we can have more records in a relation than space available in a record, we can store records in several pages. DBMS tend to save only records for the same relation in a table. There are two ways we can save records across multiple pages (Fig. 7.6):

- **Unspanned.** If each record is not allowed to be saved in multiple pages.
- **Spanned.** If the record can be divided within several pages. This uses all the space within the page. Saves more space but everytime we need a record that is saved in multiple pages, we will need to load several pages.



(a) Unspanned



(b) Spanned

Figura 7.6: Saving records across multiple pages

We can calculate the number of records that fit in a page with the blocking factor (bfr).

$$bfr = \lfloor \frac{B}{R} \rfloor$$

- B is the number of bytes for a page. This is fixed length (i.e., all pages have the same size).
- R is the number of bytes for a record.
- bfr is the blocking factor. This details the number of records that fit within a page.

We can also calculate the number of pages needed to save a number of records.

$$b = \lceil \frac{r}{bfr} \rceil$$

- r is the number of records to save.
- bfr is the blocking factor. bfr for variable-length records represents the average number of records per page.
- b is the number of pages needed to save all the records.

With regards to searching:

- If the records are ordered by attributes and we are searching for them, we can need to retrieve $O(logb)$ pages (binary search).
- If not, we must iterate over every page, in worst case, to find the record. On average, we will have to iterate over $O(n/2)$ pages.

We can see the file, page and slot (i.e., record) information in SQL Server with the following command for a table T.

```
SELECT sys.fn_PhysLocFormatter(%%physloc%%) AS [File:Page:Slot], *
FROM T;
```

7.6. Indexes

Though files and pages are stored in a certain order physically, databases use an auxiliary data structure called an *index* to provide a faster way to find the *secondary access paths* of the data within a database without altering the physical structure. Based on an *indexing field* or *indexing attribute*, the index is created to enable fast access on those fields. The DBMS ensures that the tables and indexes are synchronized. There is a trade-off between speed and additional resources required (additional space and more maintenance effort for synchronization). There are two levels of indexes: ordered indexes and multi-level indexes.

7.7. Ordered indexes

Ordered indexes are auxiliary hash tables that use the sorted order of the records and a search key to find the associated records. This is similar to a library, where books are sorted in some order (e.g., genre, author name, DOI) and we have a key with which we can search for the books based on the sorting. Thus, every record has an *index record* that saves both the search key value and the pointers to the records.

	Physically ordered by indexing key	**Physically not ordered by indexing key**
Index field is key	Primary index	Secondary index (key)
Index field is non-key	Clustering index	Secondary index(non-key)

Cuadro 7.1: Different types of indexes based on the storage order and indexing field

There are several types of ordered indexes described in the following subsections. The relationship between these is shown in Table 7.1.

Depending on the type index field, we may have an entry for every search key value (*dense index*) or entries for some search key values (*sparce or non-dense index*). Primary and clustering indexes are nondense. Secondary indexes are dense for keys, but may be dense or nondense depending on the sorting order of the records for non-key attributes.

Primary index

Records *are sorted with the index record*. The index record uses as a search key value the *key attribute* of a relation. There is a search key for the *first* record within a page (the *anchor record*). Thus, there is one key per number of pages in the ordered data files. The structure is shown in Fig. 7.7.

Clustering index

Records *are sorted with the index record*. The index record uses as a search key value a *non-key attribute* of a relation. As there can be repeated values, the index saves an entry for *each distinct* value for the index record, pointing at the page where the first record is found. The structure is shown in Fig. 7.8.

Secondary index

Records *are not sorted with the index record*. Either we can use as the search key value the key or non-key attributes of a relation. Therefore, there needs to be an index record for every value. The structure is shown in Fig. 7.9.

7.8. Multi-level indexes

Ordered indexes disadvantage is that the performance of index lookups and sequential scans does not scale well when the file grows. Furthermore, frequent reorganization of records within the file are needed but are undesirable.

Thus, *balanced tree structures* are used to maintain efficiency in insertions and deletions. In a balanced tree of n children, every non-leaf node has between $\lceil \frac{n}{2} \rceil$ and n children. Therefore the distance from the root of the tree to any leaf is the same. Specifically, *B+ Trees* are the most used auxiliary data structures for databases. However, this data structure, compared to ordered indexes, has an additional overhead for insertions, deletions and space.

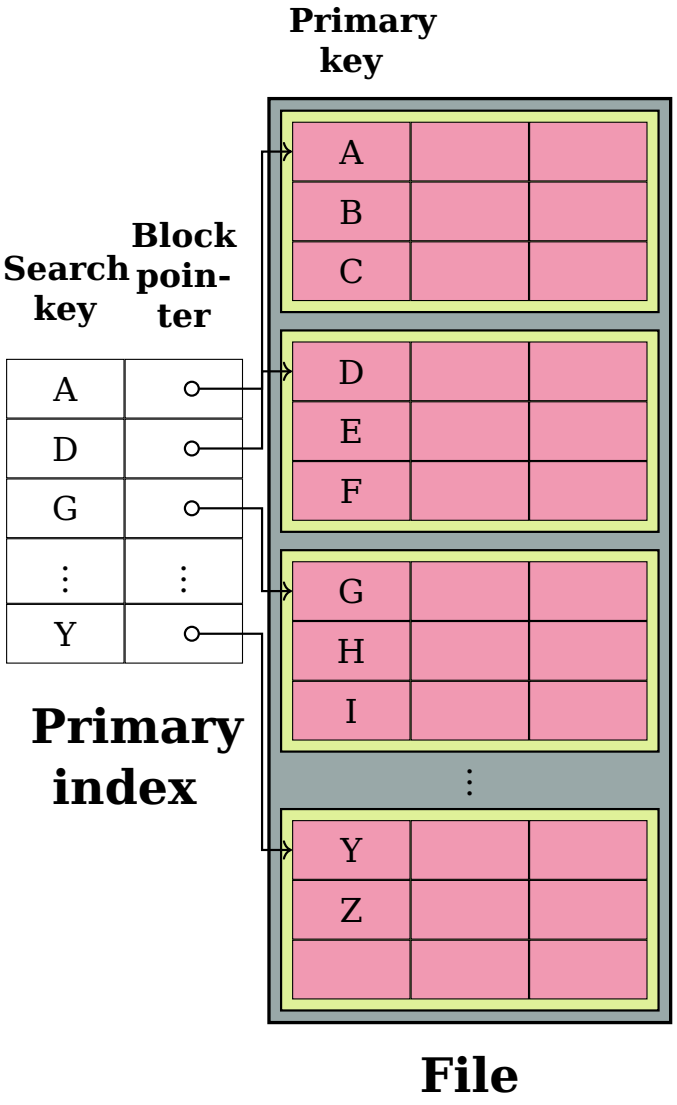


Figura 7.7: Primary index

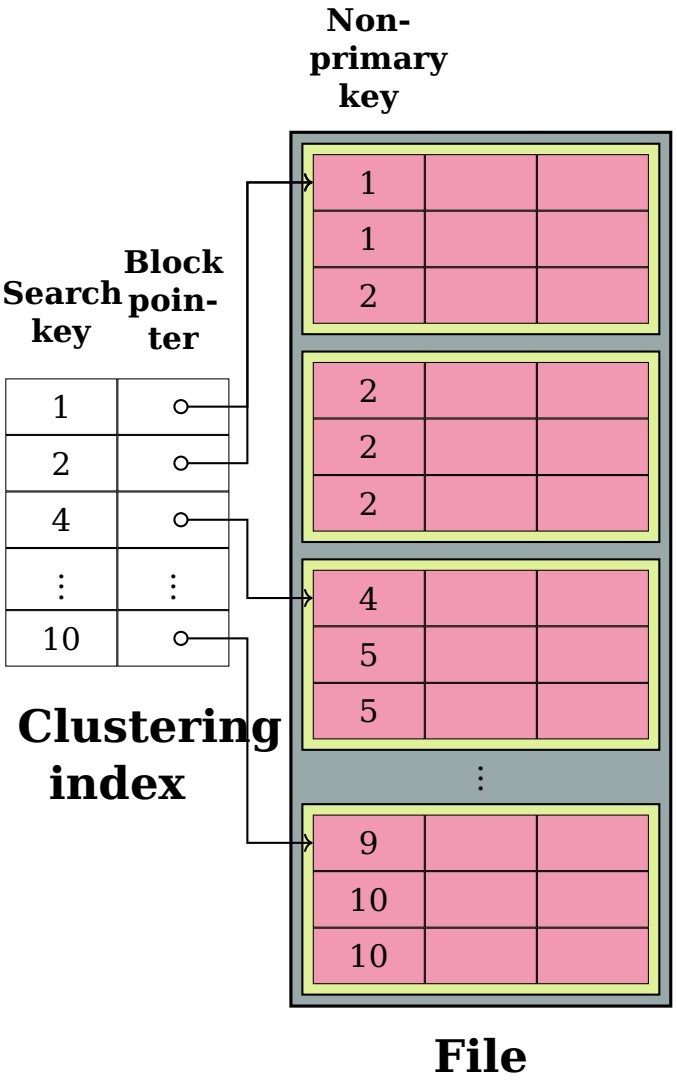


Figura 7.8: Clustering index

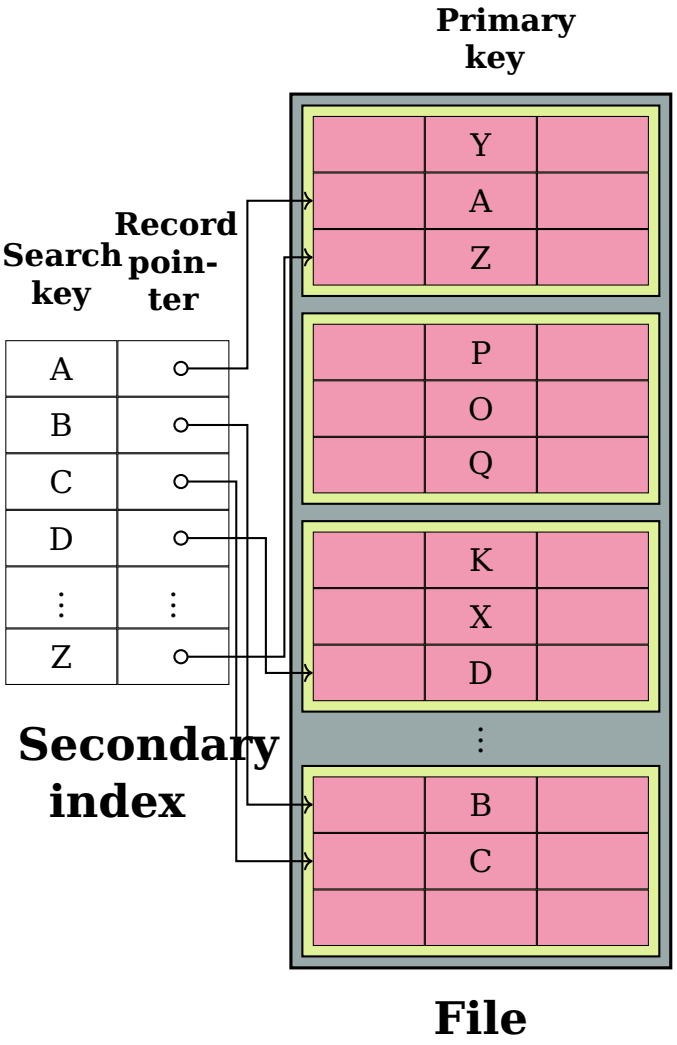


Figura 7.9: Secondary index for a primary key

7.9. B+ Tree

A *B+Tree* is a self-balancing tree that allows insertions and deletions in $log_n(b)$ time. We can see an example in Fig. 7.10. This is a tree with $n = 2$ keys. Thus every node has $n + 1 = 3$ references. Let us define that every node of the tree is a combination of references (rf) and values (v). In our example, it has the form $(rf_1, v_1, rf_2, v_2, rf_3)$.

There is only one inner node for this tree (in this example, the node with keys 5 and 9). Inner nodes are non-leaf nodes. Meanwhile, all the nodes in the lowest level are leaf nodes. There is a leaf node for every value of the search field with a pointer to either the page or record where the value of the node is stored. In this example, we are pointing to every record. The last pointer in a leaf node saves the direction where the following leaf node of the tree is.

The inner nodes have the following properties.

- Any value in the subtree in which rf_1 points to is denominated X_1. Thus, $X_1 \leq V_1$.
- Any value in the subtree in which rf_2 points to is denominated X_2. Thus, $V_1 < X_2 \leq V_2$.
- Any value in the subtree in which rf_2 points to is denominated X_3. Thus, $V_2 < X_3$.

Therefore, a B+ tree has the following properties.

- **Key-pointer property.** Every value within the tree has the key with a pointer.
- **Key-order property.** Every node keys are ordered.
- **Half-full nodes property.** Every non root inner node is at least half full ($\lceil \frac{n+1}{2} \rceil - 1 \leq numberofkeys \leq n$)
- **Balanced property.** Every leaf node has the same depth, thus it is perfectly balanced.

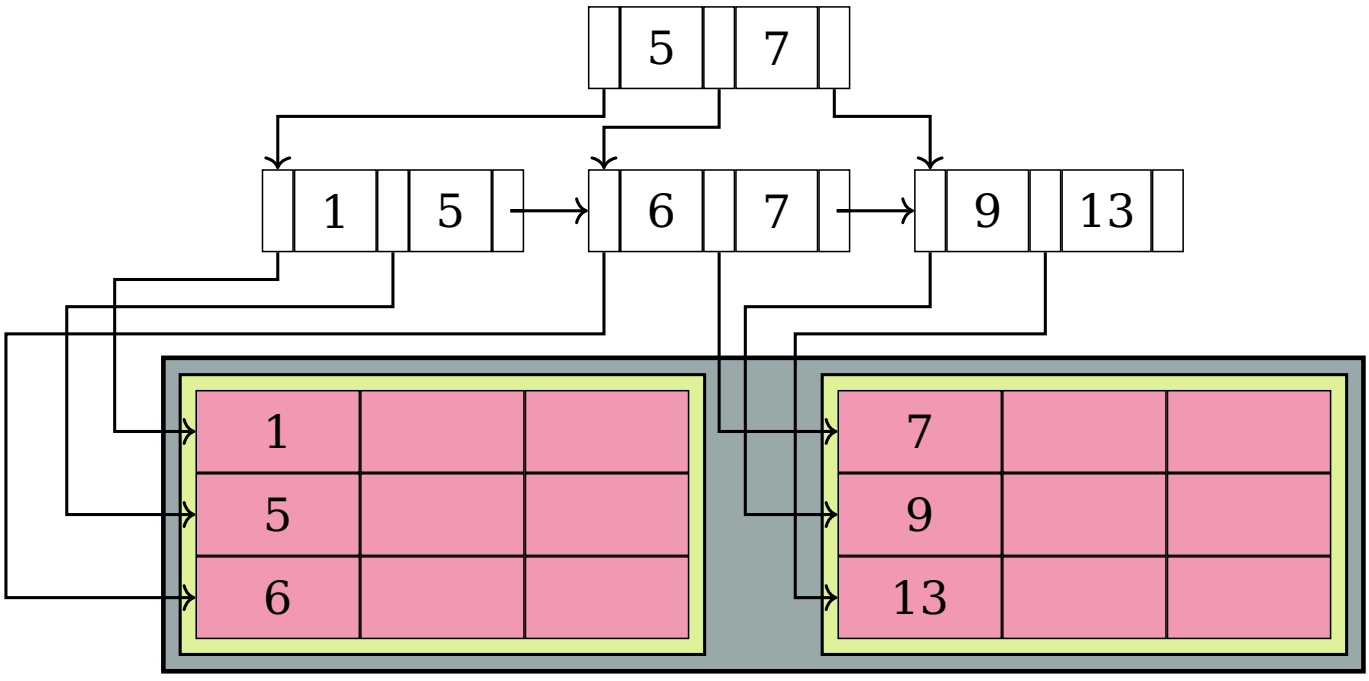


Figura 7.10: B+Tree example

Searching

We can take advantage of the structure of the tree to iterate through from the leaf to the root to find specific values. We can see the steps in Algorithm 1.

Inserting

We can also insert new values into the three. Insertions must ensure that the tree is balanced and still complies with all the conditions of the data structure. We can see the high-level steps in Algorithm 2.

Deleting

Deletions have the same restrictions as insertions. We can see the high-level steps in Algorithm 3.

Algorithm 1 Searching for records that has a search key value v of a B+ Tree of order p

---

function find(v)
  $n = tree.getRootPage()$;
  $n.readPage()$;
  $p = n.getNumberPointers()$;
  while ($!n.isLeafNode()$) **do**
    $l = n.getLarger(v)$; ▷ The node n with i x_i values, get those that are $v \leq x_i$
    $xi, pi = l.min()$; ▷ Finds the smallest x_i that is larger than v with the respective pointer p_i
    if ($xi.isNull()$) **then** ▷ $v > x_i$
      $n = n.getLastPointer()$; ▷ Gets last non-null pointer of n
    else if ($v == xi$) **then** ▷ $v = x_i$
      $n = n.getNext(pi)$; ▷ Gets the pointer following p_i
    else ▷ $v < x_i$
      $n = pi$; ▷ Gets the current pointer
    end if
    $n.readPage()$;
  end while
  $r = n.hasRecordWithKey(v)$
  if ($!r.isNull()$) **then** ▷ We did found a record with the value
    return n;
  else ▷ We did not found a record with the value
    return $null$;
  end if
end function

Algorithm 2 Inserting a record that has a search key value v of a B+ Tree of order p

function insert(v)
    $l = tree.find(v)$ ▷ We use a variation of the $find(v)$ method
    if $(l.hasSpace())$ **then**
        $l.insertOrder(v)$; ▷ Also updates the references
    else ▷ We must split the node
        $m = l.getMiddleKey()$;
        $l, l2 = l.splitEvenly()$;
        $l.parent.insertOrder(m, l2)$;
    end if
end function

Algorithm 3 Deleting a record that has a search key value v of a B+ Tree of order p

function delete(v)
    $l = tree.find(v)$ ▷ We use a variation of the $find(v)$ method
    if $(l.hasRecordWithKey(v))$ **then**
        $l.remove(v)$
        if $l.sibling.canRedistribute()$ **then** ▷ Siblings are adjacent nodes with the same parent
            $l.redistributeSibling()$;
        else
            $n, o = merge(l, l.siblings)$; ▷ n is the new merged node, while o is the old node
            $n.parent.removeReference(o)$;
        end if
    end if
end function

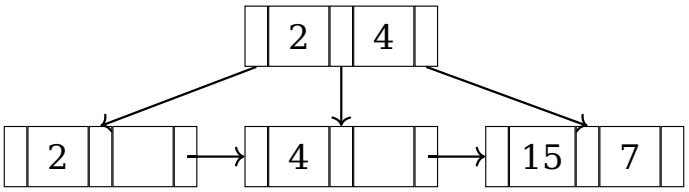
7.10. Review

Teoría

1. ¿Cuál es la diferencia entre memoría primaria, secundaria y terciaria?
2. ¿Qué tipo de memoría utiliza las bases de datos?
3. ¿Cuáles son los componentes del diseño físico de la base de datos? Indique cómo es que se relaciona cada uno.
4. ¿Por qué se utiliza un diseño basado en disco para el almacenamiento físico de las bases de datos?
5. ¿Qué es un archivo en bases de datos?
6. ¿Qué es una página en bases de datos?
7. ¿Qué es un registro en bases de datos?
8. ¿Cuál es la diferencia entre registros de longitud fija y variable?
9. ¿Cuál es la diferencia entre registros *unspanned* y *spanned*?
10. ¿Cuáles estructuras auxiliares de bases de datos existen? Explique el funcionamiento de cada uno.
11. ¿Cuál es la diferencia entre índices ordenados y multi nivel?
12. ¿Qué es un árbol B+?
13. ¿Para qué que se utiliza un árbol B+ en bases de datos?
14. ¿Por qué se utilizan estructuras auxiliares para el almacenamiento físico en las bases de datos?

Árboles B+: Errores

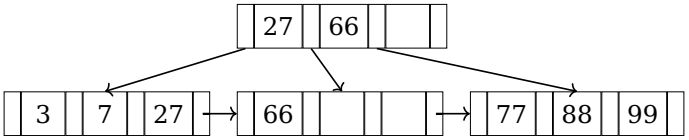
1. Se tiene el siguiente árbol B+ de $n = 2$ llaves.



¿Cuál de las siguientes propiedades de un árbol B+ incumple?

◯ *Key-pointer property* ◯ Key-order property ◯ *Half-full nodes property*
◯ *Balanced property* ◯ Ninguno

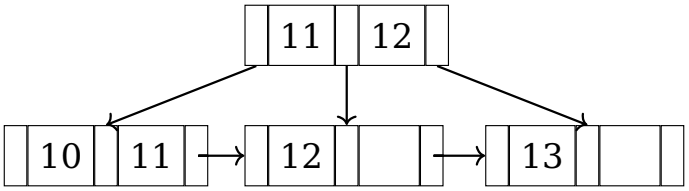
2. Se tiene el siguiente árbol B+ de $n = 3$ llaves.



¿Cuál de las siguientes propiedades de un árbol B+ incumple?

◯ *Key-pointer property* ◯ *Key-order property* ◯ *Half-full nodes property*
◯ *Balanced property* ◯ Ninguno

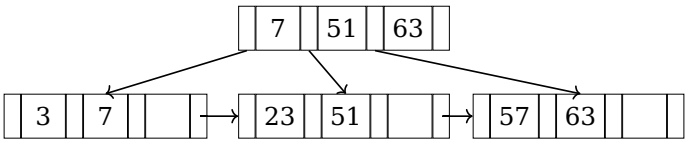
3. Se tiene el siguiente árbol B+ de $n = 2$ llaves.



¿Cuál de las siguientes propiedades de un árbol B+ incumple?

◯ *Key-pointer property* ◯ *Key-order property* ◯ *Half-full nodes property*
◯ *Balanced property* ◯ Ninguno

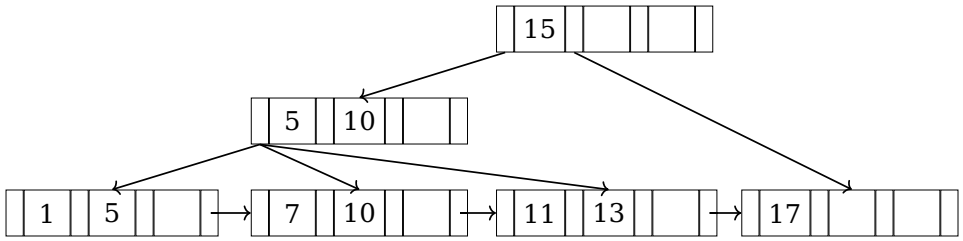
4. Se tiene el siguiente árbol B+ de $n = 3$ llaves.



¿Cuál de las siguientes propiedades de un árbol B+ incumple?

◯ *Key-pointer property* ◯ *Key-order property* ◯ *Half-full nodes property*
◯ *Balanced property* ◯ Ninguno

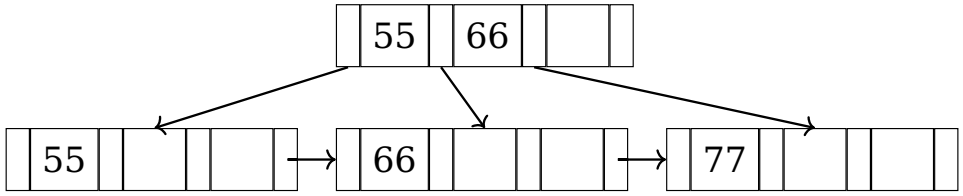
5. Se tiene el siguiente árbol B+ de $n = 3$ llaves.



¿Cuál de las siguientes propiedades de un árbol B+ incumple?

◯ *Key-pointer property* ◯ *Key-order property* ◯ *Half-full nodes property*
◯ *Balanced property* ◯ Ninguno

6. Se tiene el siguiente árbol B+ de $n = 3$ llaves.



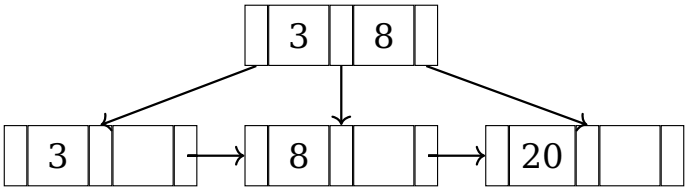
¿Cuál de las siguientes propiedades de un árbol B+ incumple?

◯ *Key-pointer property* ◯ *Key-order property* ◯ *Half-full nodes property*
◯ *Balanced property* ◯ Ninguno

Árboles B+: Algortimos de inserción y borrado

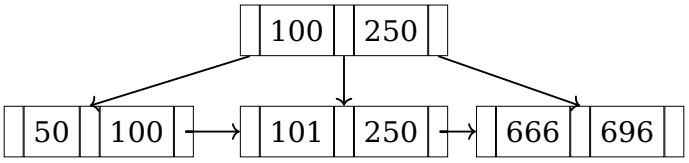
Para cada uno de los siguientes ejercicios, dibuje el árbol despues de realizar la operación con su algortimo respectivo. Para cada ejercicio, aplique cada operación en orden (una despues de otra).

1. Se tiene el siguiente árbol B+ de $n = 2$ llaves.



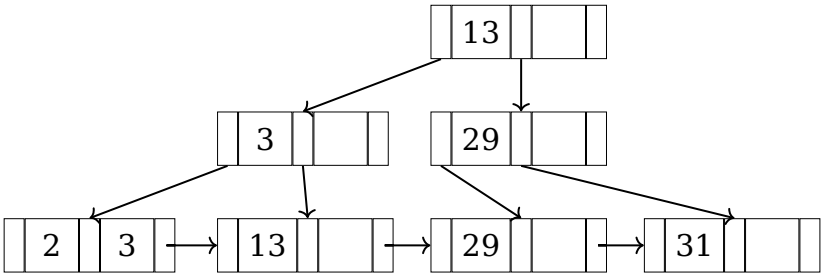
   (a) Agregue al árbol B+ el valor 4.
   (b) Agregue al árbol B+ el valor 33.
   (c) Agregue al árbol B+ el valor 5.
   (d) Remueva del árbol B+ el valor 20.
   (e) Remueva del árbol B+ el valor 3.
   (f) Remueva del árbol B+ el valor 5.

2. Se tiene el siguiente árbol B+ de $n = 2$ llaves.



   (a) Remueva del árbol B+ el valor 101.
   (b) Agregue al árbol B+ el valor 303.
   (c) Agregue al árbol B+ el valor 350.
   (d) Agregue al árbol B+ el valor 21.
   (e) Agregue al árbol B+ el valor 150.
   (f) Agregue al árbol B+ el valor 160.
   (g) Agregue al árbol B+ el valor 170.
   (h) Agregue al árbol B+ el valor 80.

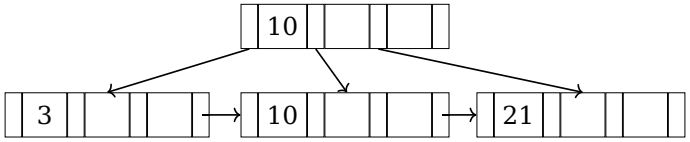
3. Se tiene el siguiente árbol B+ de $n = 2$ llaves.



   (a) Agregue al árbol B+ el valor 37.
   (b) Remueva del árbol B+ el valor 31.

(c) Agregue al árbol B+ el valor 5.
(d) Agregue al árbol B+ el valor 17.
(e) Agregue al árbol B+ el valor 43.
(f) Agregue al árbol B+ el valor 47.
(g) Agregue al árbol B+ el valor 19.
(h) Agregue al árbol B+ el valor 23.

4. Se tiene el siguiente árbol B+ de $n = 3$ llaves.



(a) Agregue al árbol B+ el valor 5.
(b) Agregue al árbol B+ el valor 18.
(c) Agregue al árbol B+ el valor 7.
(d) Agregue al árbol B+ el valor 9.
(e) Remueva del árbol B+ el valor 21.
(f) Agregue al árbol B+ el valor 12.
(g) Agregue al árbol B+ el valor 15.
(h) Remueva del árbol B+ el valor 7.
(i) Agregue al árbol B+ el valor 11.
(j) Agregue al árbol B+ el valor 13.

5. Se tiene un árbol vacio de $n = 2$ llaves.
(a) Agregue al árbol B+ el valor 1.
(b) Agregue al árbol B+ el valor 2.
(c) Agregue al árbol B+ el valor 10.
(d) Remueva del árbol B+ el valor 2.
(e) Agregue al árbol B+ el valor 7.
(f) Agregue al árbol B+ el valor 8.
(g) Agregue al árbol B+ el valor 9.
(h) Agregue al árbol B+ el valor 5.
(i) Agregue al árbol B+ el valor 3.
(j) Agregue al árbol B+ el valor 33.

6. Se tiene un árbol vacio de $n = 3$ llaves.
(a) Agregue al árbol B+ el valor 10.
(b) Agregue al árbol B+ el valor 3.
(c) Agregue al árbol B+ el valor 6.
(d) Agregue al árbol B+ el valor 1.
(e) Agregue al árbol B+ el valor 2.
(f) Agregue al árbol B+ el valor 50.

(g) Agregue al árbol B+ el valor 47.
(h) Remueva del árbol B+ el valor 6.
(i) Remueva del árbol B+ el valor 3.
(j) Agregue al árbol B+ el valor 15.
(k) Agregue al árbol B+ el valor 5.
(l) Agregue al árbol B+ el valor 9.

Capítulo 8

Álgebra relacional

8.1. Introduction

Relational algebra is a formal language that defines a set of operations to the relational model. A sequence of relational algebra operations is a *relational algebra expression*. The results of applying an expression is a new relation that represents the result of the query in the database. Relational algebra is important as: (1) it formally defines relational model operations; (2) is used in processing and optimizing queries in RDMSs; and, (3) some concepts are used in the query language for RDMSs. We can also represent relational algebra expressions as a *query tree* or *query graph*.

The operations can be classified into two groups:

Set operations . These operators are related to set theory. These include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT.

Relation operations. These operators are developed for relational databases. These include SELECT, PROJECT, and JOIN.

Furthermore, depending on the number of operators, we can also classify the operations as *unary operations* if they operate on only one relation (e.g., SELECT, PROJECT) or *binary operations* if they operation if they operate on two relations (e.g., JOIN).

8.2. SELECT operation

The SELECT operation (σ) selects all the tuples that satisfy a condition (c) from a relation (R). This operation works as a filter to the tuples (horizontal partition) that satisfy a condition with those that don't. Thus, we can say that the select operation selects certain rows. The following notation is used.

$$\sigma_c(R)$$

- The number of tuples selected will always be less or equal than the number of tuples in R.
- The fraction of tuples selected by the conditions is the *selectivity* of the condition.
- The condition can have multiple clauses separated by boolean operators (**AND** or **OR**). We can also apply a **NOT** to reverse the boolean result.

- The clauses can be between attributes ($Fname == Lname$) or between an attribute and a constant value ($Fname == `Pedro'$).
- The clauses can compare one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$. If the values compared are unordered (e.g. Strings that represent colors) only the operators can be used $\{=, \neq\}$.
- σ in SQL is the **WHERE** clause.

Example: Get all the students who are named Pedro and are born after 1990.

$$\sigma_{Fname=`Pedro' AND Byear>1990}(STUDENT)$$

```
SELECT *
FROM STUDENT
WHERE Fname = 'Pedro' AND Byear > 1990
```

8.3. PROJECT operation

The PROJECT operation (π) selects a list of attributes (L) of a relation (R). This operation works as a vertical partition between the needed columns and those not needed. Thus, we can say that the project operation selects certain columns. The following notation is used.

$$\pi_L(R)$$

- The columns are retrieved in the same order as they appear in the list.
- The number of tuples selected will always be less or equal than the number of tuples in R.
- The *degree* is equal to the number of attributes in L.
- The operation removes duplicated tuples using *duplicate elimination*.
- π in SQL is specified by the attribute list in the **SELECT** clause with a **DISTINCT**.

Example: Get the first and last name for all the students.

$$\pi_{Fname,Lname}(STUDENT)$$

```
SELECT DISTINCT Fname, Lname
FROM STUDENT
```

8.4. RENAME operation

The RENAME operator (ρ) can rename a relation or attributes names. To rename a relation R into a new relation S, the notation is the following:

$$\rho_S(R)$$

To rename attributes based on a list of attributes $L = \{A_1, \cdots, A_m\}$ of a relation R, the notation is the following.

$$\rho_{(A_1,\cdots,A_m)}(R)$$

To rename the relation and the attribute names, the notation is the following.

$$\rho_{S(A_1,\cdots,A_m)}(R)$$

- The rename is applied in order.
- ρ in SQL is the **AS** aliasing.

Example: Rename the table STUDENT, and the attributes of the first name and birth year.

$$\rho_{S(FirstName,BirthYear)}(STUDENT)$$

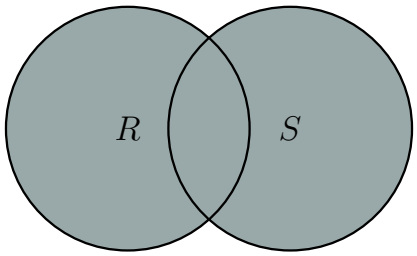
```
SELECT S.Fname AS FirstName, S.Byear AS BirthYear
FROM STUDENT AS S
```

8.5. UNION operation

The UNION operation ($\cup$) gets all the tuples in R or S, removing duplicate tuples. The following notation is used.

$$R \cup S$$

In a Venn Diagram, the UNION operation looks as following:



- To apply the operator, the *union compilability* or *type compatibility* condition must be followed. The attributes of the relations must have the same degree (number of attributes) and the corresponding pairs of attributes the same domain. Thus, the type of attributes must be the same.
- $\cup$ in SQL is **UNION**. **UNION ALL** does not eliminate duplicates.

Example: Get the list of all students or instructors.

$$STUDENT \cup INSTRUCTOR$$

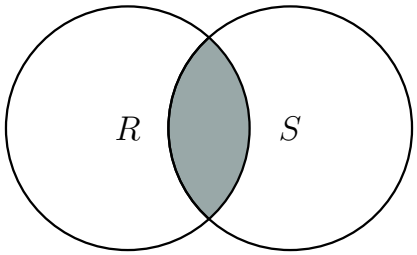
```
SELECT *
FROM STUDENT
UNION
SELECT *
FROM INSTRUCTOR
```

8.6. INTERSECTION operation

The INTERSECTION operation ($\cap$) gets all the tuples in R and S. The following notation is used.

$$R \cap S$$

In a Venn Diagram the operation is the following:



- The *type compatibility* condition applies for the operator.
- $\cap$ in SQL is **INTERSECTION**. **INTERSECTION ALL** does not eliminate duplicates.

Example: Get the list of people who are both students and instructors.

$$STUDENT \cap INSTRUCTOR$$

```
SELECT *
FROM STUDENT
INTERSECTION
SELECT *
FROM INSTRUCTOR
```

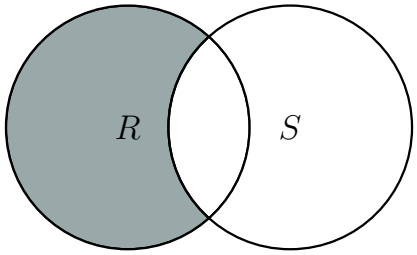
8.7. MINUS operation

The MINUS operation ($-$) gets all the tuples in R but not in S. The following notation is used.

$$R - S$$

In a Venn Diagram the operation is the following:

- It is also called *SET DIFFERENCE* or *EXCEPT*.
- The *type compatibility* condition applies for the operator.
- $-$ in SQL is **EXCEPT**. **EXCEPT ALL** does not eliminate duplicates.



Example: Get the list of students who are not instructors.

$$STUDENT - INSTRUCTOR$$

```
SELECT *
FROM STUDENT
EXCEPT
SELECT *
FROM INSTRUCTOR
```

8.8. CARTESIAN PRODUCT operation

The CARTESIAN PRODUCT operation ($\times$) combines every tuple from set $R(A_1, ..., A_n)$ with every tuple in set $S(B_1, ..., B_m)$. The following notation is used.

$$R \times S$$

- It is also called *CROSS PRODUCT* or *CROSS JOIN*.
- The number of tuples created will be $n_R * n_S$. n_R is the number of tuples in R, while n_S is the number of tuples in S.
- The degree of attributes of a tuple is $n + m$. n is the degree in R, while n_S m is the degree in S.
- The tuples for the new relation are generated in the order $(A_1, ..., A_n, B_1, ..., B_m)$.
- We mostly apply this operation, followed by a σ operation.
- $\times$ in SQL is **CROSS JOIN**. Also, if there are two tables in the **FROM** clause without a join condition in the **WHERE** clause.

Example: Get all the combinations of students enrolled to a course.

$$STUDENT \times ENROLLED$$

```
SELECT *
FROM STUDENT, ENROLLED
```

or

```
SELECT *
FROM STUDENT CROSS JOIN ENROLLED
```

8.9. JOIN operation

The JOIN operation ($\bowtie$) combines two relations (R and S) based on condition c. Only the combined tuples that satisfy the condition c are included in the resulting relation as a combined tuple. The following notation is used.

$$R \bowtie_c S$$

- We can have multiple conditions unified by boolean operators in c.
- A *THETA JOIN* has one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$ denominated θ.
- An *EQUIJOIN* uses only the $=$ comparison operator. The columns involved in the *EQUIJOIN* are repeated, as both have the same values in each column.
- A *NATURAL JOIN* ($*$) is an *EQUIJOIN* that gets rid of the repeated column by combining both relations by a join attribute. The join attribute has the same name in both relations. If there is no column with the same name, it can be renamed using the ρ operation.
- Without a condition c, the *JOIN* becomes a *CARTESIAN PRODUCT*.
- The number of tuples created will be between 0 and $n_R * n_S$. n_R is the number of tuples in R, while n_S is the number of tuples in S.
- The degree and order of the resulting attributes is the same as the *CARTESIAN PRODUCT*.
- The *join selectivity* is the expected size of the join result divided by the maximum size of $n_R * n_S$.
- The *JOIN*s are *inner joins*.
- $\bowtie$ can be applied in SQL in various ways: (1) by specifying the join condition in the **WHERE** clause; (2) using a nested relation; and, (3) using join table instructions such as **JOIN**, **NATURAL JOIN** or **LEFT OUTER JOIN**.

Example: Get the courses students have enrolled.

$$(\rho_S(STUDENT)) \bowtie_{S.id=E.id} (\rho_E(ENROLLED))$$

```
SELECT *
FROM STUDENT AS S, ENROLLED AS E
WHERE S.id = E.id
```

or

$$STUDENT * ENROLLED$$

```
SELECT *
FROM STUDENT AS S JOIN ENROLLED AS E ON S.id = E.id
```

```
SELECT *
FROM STUDENT NATURAL JOIN ENRO-
LLED
```

8.10. Sequences of operations

Generally, we apply several algebraic operations for a query. We could write an *in-line expression* that has all the operations in an expression. Alternatively, we can save in an intermediate relation the result using the assignment operator ($\leftarrow$).

Example: Get the name of the students born after 1990.

The in-line expression is:

$$\pi_{Fname,Lname}(\sigma_{Byear>1990}(STUDENT))$$

Using intermediate relations is:

$$OVER1990 \leftarrow \sigma_{Byear>1990}(STUDENT)$$
$$RESULT \leftarrow \pi_{Fname,Lname}(OVER1990)$$

8.11. Complete set of operations

With the operations $\{\sigma, \pi, \cup, \rho, -, \times\}$ we can generate all the other algebra operations as a sequence of operations. Thus, these operations are a *complete set*. We can derive the following operations, though they are useful to express as they are frequently applied in DMS.

- $R \cap S \equiv (R \cup S) - ((R \cup S) \cup (S \cup R))$
- $\sigma_c(R \times S) \equiv (R \bowtie_c S)$
- $R \div S \equiv \pi_y(R) - \pi_y((S \times \pi_y(R)) - R)$

8.12. Equivalences

There are many rules that can transform a relational algebra expression into an equivalent expression. Two relational algebra expressions are *equivalent* if they have the same set of attributes in a different order but the expressions represent the same information. Thus, two expressions are equivalent if the same set of tuples is retrieved.

1. **Cascade of σ.** A selection condition with AND can be broken up into a sequence of individual σ operations.

$$\sigma_{c_1 AND c_2 AND \cdots AND c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\cdots(\sigma_{c_n}(R))\cdots))$$

Example: Get the students with the first name Pedro, born after 1990.

$$\sigma_{Fname=`Pedro' AND Byear>1990}(STUDENT) \equiv \sigma_{Fname=`Pedro'}(\sigma_{Byear>1990}(STUDENT))$$

2. **Commutativity of σ.** σ can be applied in any order.

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

Example: Get the students with the first name Pedro, born after 1990.

$$\sigma_{Byear>1990}(\sigma_{Fname=`Pedro'}(STUDENT) \equiv \sigma_{Fname=`Pedro'}(\sigma_{Byear>1990}(STUDENT))$$

3. **Cascade of π.** When there is a sequence of π operations, all except the outermost one can be ignored.

$$\pi_{List_1}(\pi_{List_2}(\cdots(\pi_{List_n}(R))\cdots)) \equiv \pi_{List_1}(R)$$

Example: Get the if of the students.

$$\pi_{id}(\pi_{id,Fname}(\pi_{id,Fname,Lname}(STUDENT))) \equiv \pi_{id}(STUDENT)$$

4. **Commuting σ with π.** If the selection condition c involves only attributes $A_1, A_2, \cdots, A_N$ in the projection list, then the two operations can be applied in any order.

$$\pi_{A_1,A_2,\cdots,A_N}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1,A_2,\cdots,A_N}(R))$$

Example: Get the id with the full name of students with the first name Pedro.

$$\pi_{id,Fname,Lname}(\sigma_{Fname=`Pedro'}(STUDENT)) \equiv \sigma_{Fname=`Pedro'}(\pi_{id,Fname,Lname}(STUDENT))$$

5. **Commutativity $\bowtie$ and $\times$.** $\bowtie$ and $\times$ can be applied in any order.

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

Example: Get the courses students have enrolled.

$$STUDENT \bowtie_{S.id=E.id} ENROLLED \equiv ENROLLED \bowtie_{S.id=E.id} STUDENT$$

6. **Commutativity of σ with $\bowtie$ and $\times$.** If all the attributes of the selection condition c involve only the attributes of the joined relations, they can be applied in any order.

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$$

Example: Get the students with the first name Pedro and born after 1990, with their course enrollment information.

$$\sigma_{Fname=`Pedro'ANDByear>1990}(STUDENT \bowtie_{S.id=E.id} ENROLLED) \equiv$$
$$(\sigma_{Fname=`Pedro'ANDByear>1990}(STUDENT)) \bowtie_{S.id=E.id} ENROLLED$$

Alternatively, if c can be written as c_1 and c_2 where c_1 has only the attributes of R and c_2 has only the attributes of S, they can be applied in any order.

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$$

Example: Get the students with the first name Pedro and with their enrolled courses where they got grades higher than 90.

$$\sigma_{Fname=`Pedro'ANDgrade>90}(STUDENT \bowtie_{S.id=E.id} ENROLLED) \equiv$$
$$(\sigma_{Fname=`Pedro'}(STUDENT)) \bowtie_{S.id=E.id} (\sigma_{grade>90}(ENROLLED))$$

7. **Commutativity of π with $\bowtie$ and $\times$.** For a list of attributes of the projection is called L, such that $L = A_1, \cdots, A_n, B_1, \cdots, B_m$ where $A_1, \cdots, A_n$ are attributes of R and $B_1, \cdots, B_n$ are attributes of S.

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1,\cdots,A_n}(R)) \bowtie_c (\pi_{B_1,\cdots,B_m}(S))$$

If attributes $A_{n+1}, \cdots, A_{n+k}$ of R and $B_{m+1}, \cdots, B_{m+p}$ of S are involved for the join condition but are not in L, then these attributes must be added to the join condition. This does not apply for the $\times$ operator.

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1,\cdots,A_n,A_{n+1},\cdots,A_{n+k}}(R)) \bowtie_c (\pi_{B_1,\cdots,B_m,B_{m+1},\cdots,B_{m+p}}(S))$$

Example: Get the name and course initials of students enrolled in courses.

$$\sigma_{Fname,Lname,Sigla}(STUDENT \bowtie_{S.id=E.id} ENROLLED) \equiv$$
$$(\pi_{Fname,Lname,id}(STUDENT)) \bowtie_{S.id=E.id} (\pi_{id,Sigla}(ENROLLED))$$

8. **Commutativity of $\cap$ and $\cup$.** $\cap$ and $\cup$ can be applied in any order.

$$R \cap S \equiv S \cap R$$

$$R \cup S \equiv S \cup R$$

Example: Get the list of people who are both students and instructors.

$$STUDENT \cap INSTRUCTOR \equiv INSTRUCTOR \cap STUDENT$$

9. **Associativity of $\bowtie$, $\times$, $\cap$ and $\cup$.** These four operations, represented by a θ, then can be associated in any order.

$$(R\theta S)\theta T \equiv R\theta(S\theta T)$$

Example: Get for all the students their enrolled courses with the course information.

$$(STUDENT \bowtie_{S.id=E.id} ENROLLED) \bowtie_{E.sigla=C.sigla} COURSE \equiv$$
$$STUDENT \bowtie_{S.id=E.id} (ENROLLED \bowtie_{E.sigla=C.sigla} COURSE)$$

10. **Commutativity of σ with set operations.** These set operations $\cap$, $\cup$ and $-$ (represented by θ can be applied in any order with σ

$$\sigma_c(R\theta S) \equiv (\sigma_c(R))\theta(\sigma_c(S))$$

Example: Get the list of people who are both students and instructors named Pedro.

$$\sigma_{Fname=`Pedro'}(STUDENT \cap TEACHER) \equiv$$
$$(\sigma_{Fname=`Pedro'}(STUDENT))\theta(\sigma_{Fname=`Pedro'}(TEACHER))$$

11. **Commutativity of π with $\cup$.** π and $\cup$ can be applied in any order.

$$\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$$

Example: Get the list of all students or instructors born after 1990.

$$\pi_{Byear>1990}(STUDENT \cup TEACHER) \equiv$$
$$(\pi_{Byear>1990}(STUDENT)) \cup (\pi_{Byear>1990}(TEACHER))$$

12. **Converting a ($\sigma, \times$) sequence into $\bowtie$.** If a $\times$ is followed by a condition c of a σ, then it can be changed into a $\bowtie$.

$$\sigma_c(R \times S) \equiv (R \bowtie_c S)$$

Example: Get the courses students have enrolled.

$$\sigma_{s.id=e.id}(STUDENT \times ENROLLED) \equiv (STUDENT \bowtie_{s.id=e.id} ENROLLED)$$

There are other transformations such as pushing σ to set operators, removing trivial transformations and De Morgan's laws.

8.13. Query tree

A *query tree* is a tree data structure that represents a relational algebra expression. The *leaves* of the tree are the inputs. The *internal nodes* represent the relational algebra operations. When we execute a node when its operands are available and we replace the node with the executing results. The tree specified the order to execute the operations of the query. We start by executing the leaf nodes and finish when the root node is executed, producing the query result.

Example: Get the id of students who have gotten a 90 in an enrolled course.

$$\pi_{S.id}(\sigma_{E.grade=90}(STUDENT \bowtie_{S.id=E.id} ENROLLED))$$

```
SELECT S.id
FROM STUDENT as S, ENROLLED
as E
WHERE S.id = E.id AND E.grade =
90
```

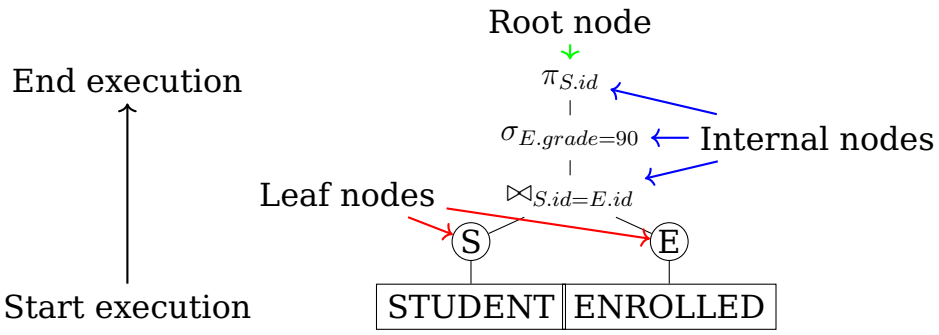


Figura 8.1: Parts of a query tree

8.14. Query graph

A *query graph* is a data structure that represents a relational calculus expression. Relationships are expressed as *relation nodes* with circles. Constant values from the query selection condition, represented by double circles or ovals, are represented by *constant nodes*. Selection or join operations are represented by *edges*. Finally, the *attributes* retrieved per relation are displayed in square brackets above each relation. Query graphs have no order, therefore there is only one query graph per query.

Example: Get the id of students who have gotten a 90 in an enrolled course.

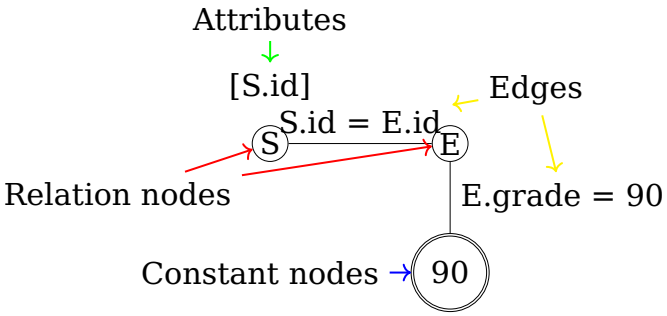


Figura 8.2: Parts of a query graph

Capítulo 9

Procesamiento de queries

9.1. Processing a query

DBMS are high-level query languages that are declarative in nature. High-level query languages indicate *what* results are expected but not *how* to get them. Meanwhile, in lower-level navigation languages (e.g., DML or the hierarchical DL/1), the programmer indicates *how* the query is executed. Thus, the query needs to be processed from a high-level query into an executable query plan.

The query execution process has the following steps shown in Fig. 9.1. An example of the inputs and outputs of each step can be seen in Fig. 9.2. In the following paragraphs, each step is described:

Requests a query. First, a query is requested from the application. The request is then received by the database to be processed and executed.

Validates the query. The query is validated syntactically and semantically. First, the query uses a *scanner* to identify the keywords, attributes name and relationships names. Then, the syntax is validated with a *parser*. Finally, the attributes and relationship names are validated by checking if the names being queried are related to the particular database queried. The result of this step is the generation of *query tree* or *query graph* that represents the query.

Selects the query. Using the representation of the query, the database must devise a strategy or plan to recover the information. Choosing the strategy is called *query optimization*. Not always the best strategy is selected, as it could be time consuming to calculate, thus it is more accurate to say that a good strategy is selected. There are two strategies for optimization, either heuristics (Section 9.3) or estimation (Section 9.5).

Generates the code. From the good execution plan, code is generated so it can be executed by a *code generator*.

Executes query code. The code is executed in a *runtime database processor* in either compiled or interpreted mode. The result is the tuples that satisfy the query or the error message.

9.2. Translating SQL queries to relational algebra

A SQL query from a RDMS is translated to the equivalent extended algebra expression, represented as a query tree. We can decompose every query into *query blocks* that contains a single **SELECT-FROM-WHERE-GROUP BY-HAVING** expression.

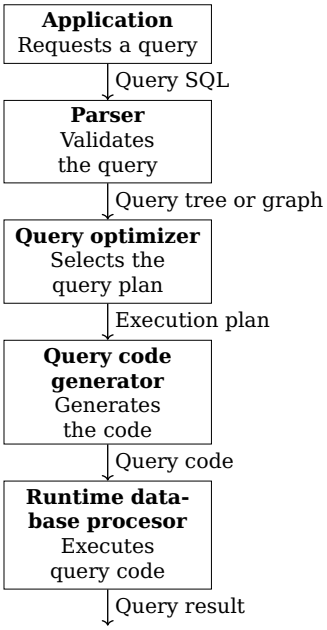


Figura 9.1: High-level query processing

- A *nested subquery block* occurs when the result of an inner query is used by an outer block.
- A *correlated nested subquery* occurs when a tuple variable from an outer block is used in the **WHERE** clause of the inner block. DBMSs use many techniques to unnest these queries.

Example: Get the name of the students with a higher admission grade than the student with id B66666.

Is separated into two query blocks:

- **Inner block.** Get the admission grade of the student with id B66666.

$$\pi_{AdmissionGrade}(\sigma_{id='B66666'}(STUDENT))$$

- **Outer block.** Get the name of the students with a higher admission grade than c.

$$\pi_{Fname,Lname}(\sigma_{AdmissionGrade>c}(STUDENT))$$

9.3. Heuristic optimization

A *heuristic* is a rule that works well in most cases, but is not guaranteed to work. The heuristics will improve the performance of the representation of a query (query tree or a query graph). As query graphs have only one representation per query, in practice query optimizers use query trees as operations have order.

The query is decomposed into *query blocks* (Section 9.2). Then, the *canonical tree* is generated, which is the initial query tree. This tree is optimized using relational algebra equivalence rules while using heuristics that preserve the equivalence of the tree. After finishing the transformation a *final query tree* is generated.

The outline of the algorithm to produce a more efficient tree to execute (in most cases) is described in the following steps.

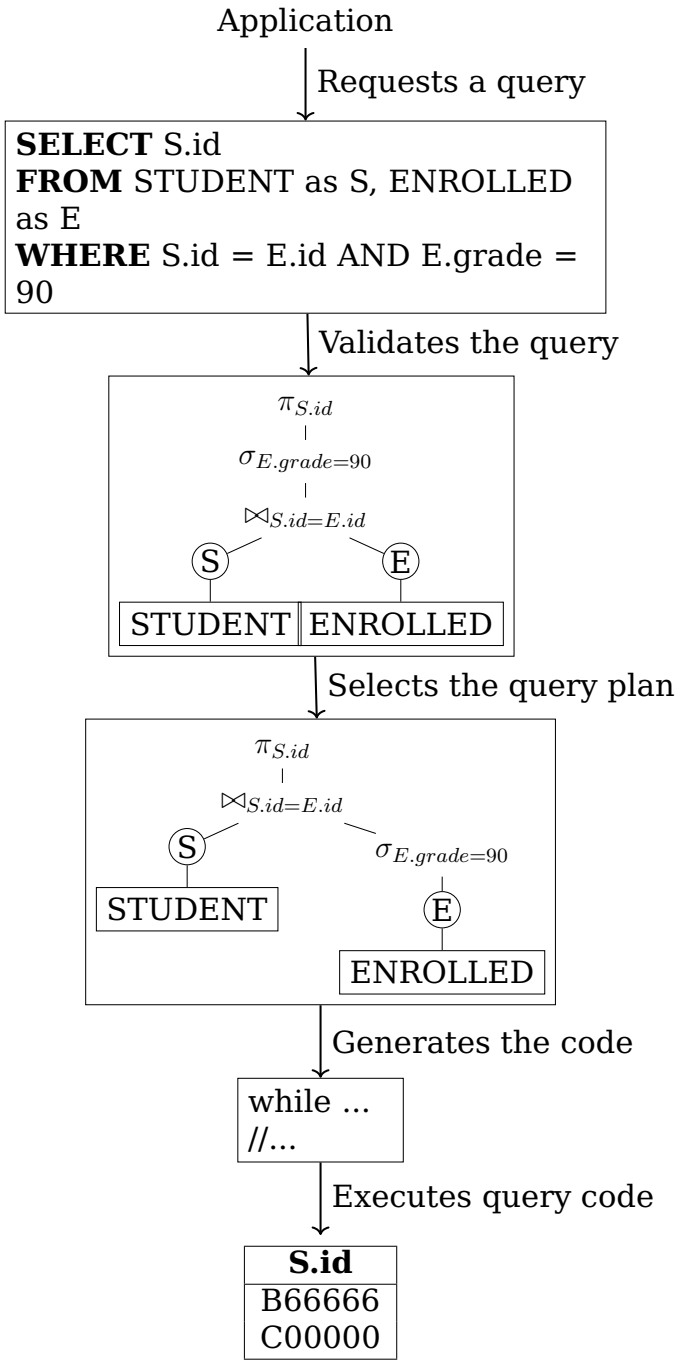


Figura 9.2: Example of the inputs and outputs of each step high-level query processing

SELECT Fname, Lname
FROM STUDENT
WHERE AdmissionGrade > (**SELECT** AdmissionGrade
FROM STUDENT
WHERE id = 'B66666');

SELECT AdmissionGrade
FROM STUDENT
WHERE id = 'B66666'

SELECT Fname, Lname
FROM STUDENT
WHERE AdmissionGrade $> c$

1. We will break up any σ operations with boolean operations into a sequence of σ operations (Cascade of σ). Separating the σ allows for a higher degree of liberty to move operations in the tree.
2. Move down σ operations as far down as it can go to be closer with the attributes of the relationship (Commutativity of σ, Commuting σ with π, Commutativity of σ with $\bowtie$ and $\times$, and Commutativity of σ with set operations).
   a) If the condition involves only one table, it means it is a *select condition* and can be moved all the way to the leaf node of the table.
   b) If it involves two tables, that means that it represents a *join condition* and can be moved above the location where the two tables are combined.
3. Rearrange the leaf nodes based on the following criteria (Commutativity $\bowtie$ and $\times$, and associativity of $\bowtie$, $\times$, $\cap$ and $\cup$).
   a) Execute first the most restrictive σ, thus move the relationship before. The most restrictive σ is the one that produces the least amount of tuples, with the smallest absolute size, or the smallest selectivity.
   b) Make sure that the order of the nodes does not cause $\times$. This may not be done if the previous relations had a select to a key field.
4. If there is $\times$ with a σ operation, combine it if it represents a $\bowtie$ condition (Converting a $(\sigma, \times)$ sequence into $\bowtie$).
5. Break down and move π as far down as possible. Create new π if needed. Only leave the π needed by each operation. (Cascade of π, Commuting σ with π, Commutativity of σ with $\bowtie$ and $\times$, and Commutativity of π with $\cup$).
6. Identify trees that can be represented by a single algorithm.

Example: Get the student IDs of those who have taken the 'CI-0127' course born after 1990

SELECT S.id
FROM STUDENT as S, ENROLLED as E, COURSE as C
WHERE C.sigla = 'CI-0127' AND S.id = E.id AND C.sigla = E.sigla AND S.Byear $>$ 1990

$$\pi_{S.id}(\sigma_{C.sigla=`CI-0127' AND S.id=E.id AND C.sigla=E.sigla AND S.Byear>1990}(S \times E \times C))$$

9.4. Query execution plan

After creating the optimized execution tree (Section 9.3), for each operation an implementation strategy is chosen. There are two approaches for executing the query:

Materialized evaluation. The result of an operation is stored as a temporary relation. Thus, it is physically materialized.

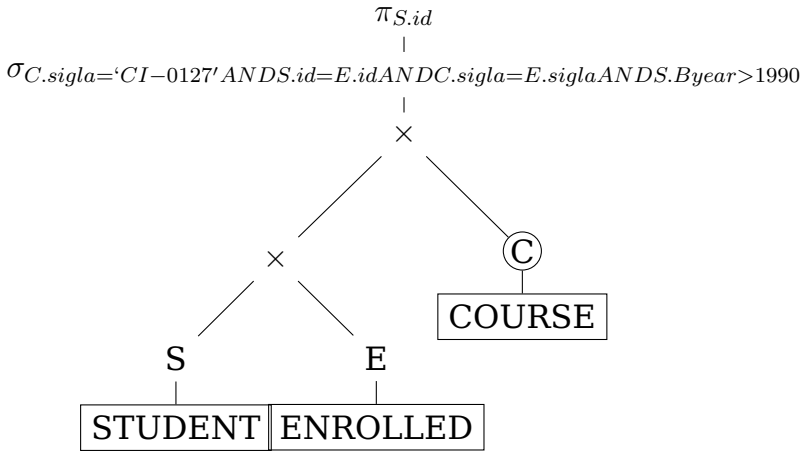


Figura 9.3: Canonical tree

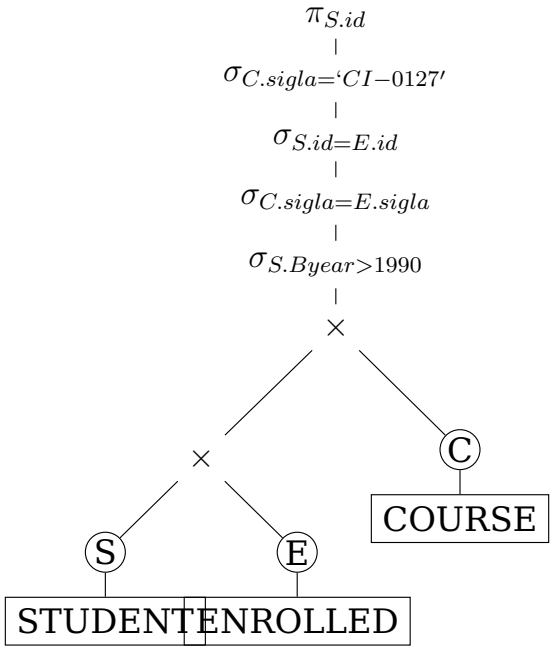


Figura 9.4: Query tree after breaking up σ operations

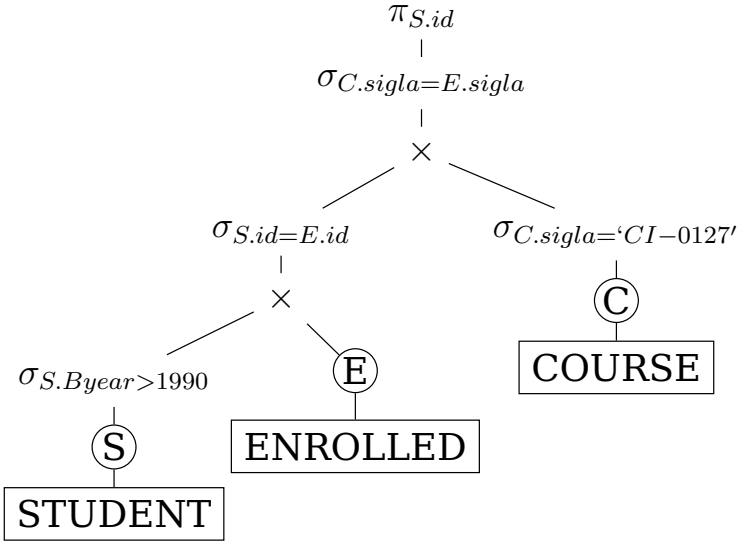


Figura 9.5: Query tree after moving down σ operations

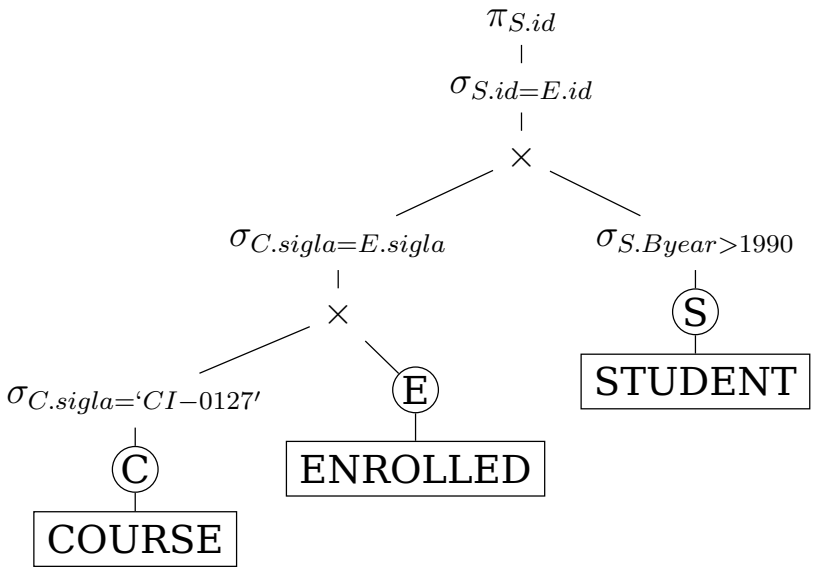


Figura 9.6: Query tree after selecting the most restrictive σ

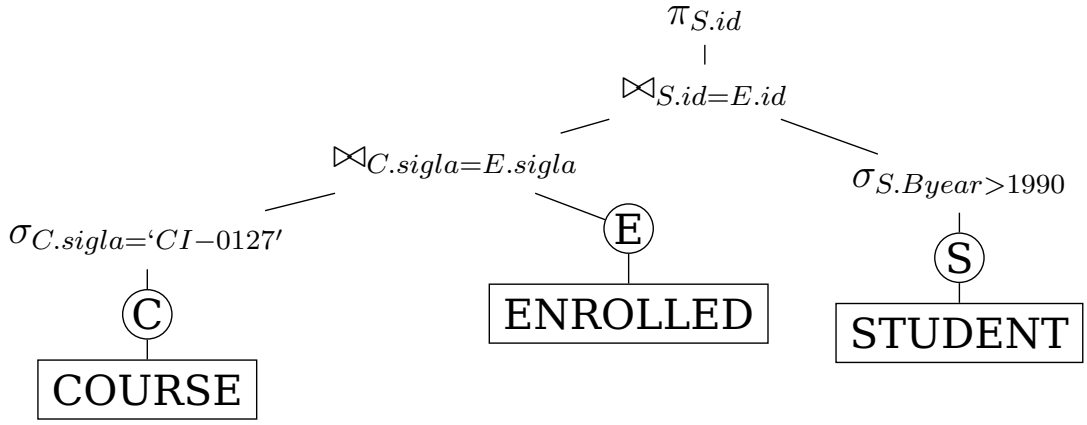


Figura 9.7: Query tree after converting to $\bowtie$

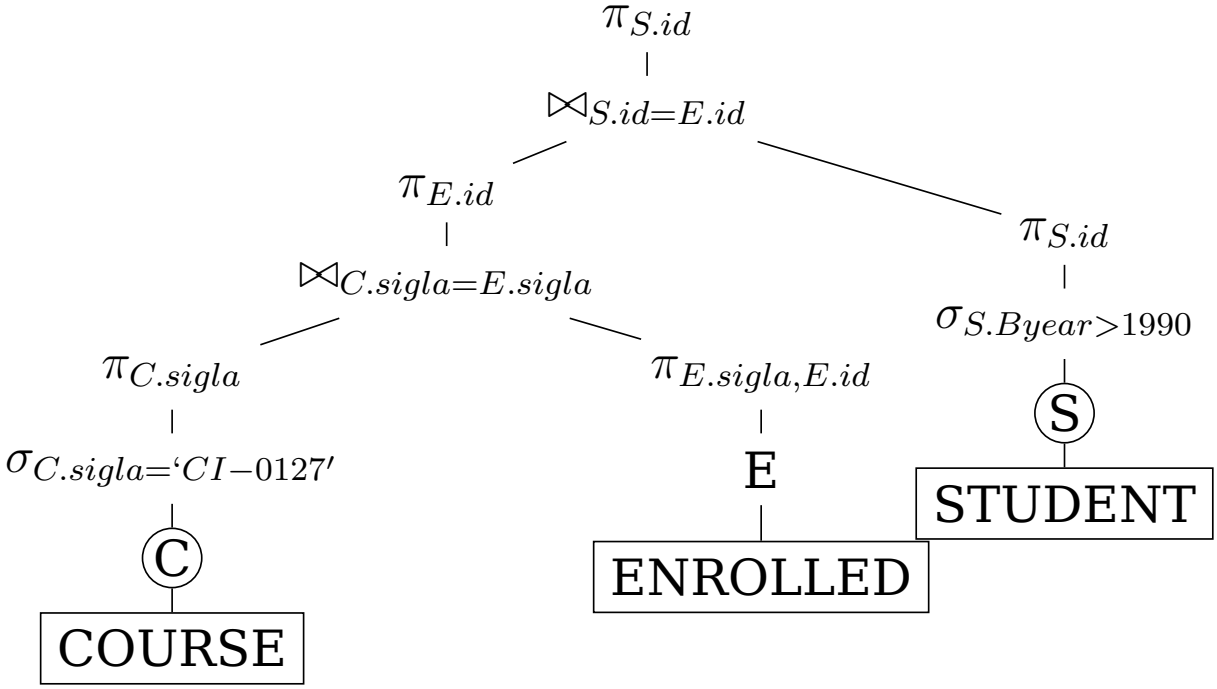


Figura 9.8: Query tree after moving down π

Pipelined evaluation. The resulting tuples are forwarded directly to the next operation through a buffer. This has an advantage in cost for writing and reading from disk.

9.5. Cost-based query optimization

Another way to optimize involves *estimating the cost* of different executing strategies and choosing the plan with the *lowest cost estimate*. Not all strategies are considered as too much time will be spent estimating. An example of different plans is shown in Fig. 9.9. Different variations of the same query are generated and estimated with the cost. Costs are also estimated for different algorithms. Based on the lowest cost, a plan is selected and executed.

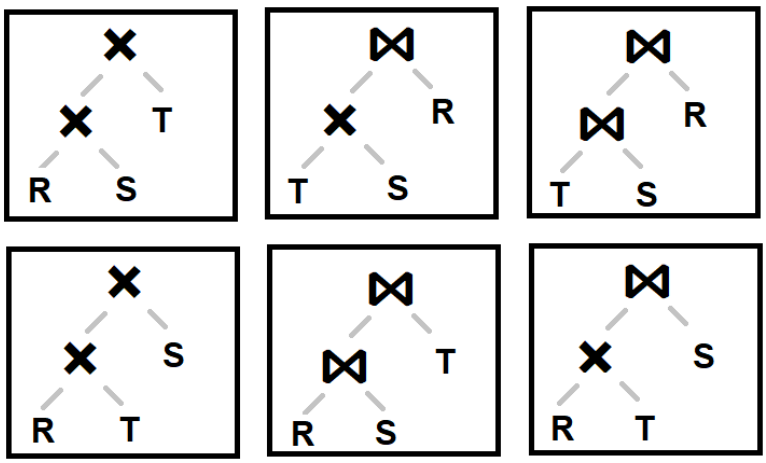


Figura 9.9: Example of generating multiple execution plans

This approach is more suited for compiled queries, optimizing at compile time, rather than interpreted queries that less time-consuming optimization works best. Traditional optimization techniques are used to search the *solution space* to find a solution that minimizes our cost functions (estimates).

The cost depends on several underlying metrics:

- **Access cost to secondary storage (a.k.a disk I/O cost).** Cost of reading and writing data blocks between main memory and secondary storage. This measures the number of block transfers. This dependinds the type of access structures and other factors such as the allocation of memory.
- **Disk storage cost.** Cost of storing in disk any intermediate files generated by an execution strategy for the query.
- **Computation cost (a.k.a CPU cost).** Cost of using the CPU such as to search, order and computing fields. This measures the CPU used, but it is tough to estimate.
- **Memory usage cost.** Cost of main memory buffers used during query execution.
- **Communication cost (a.k.a network cost).** Cost of shipping the query and the results from the database to the application that requests the query. This measures the number of messages sent.

For large databases there is an emphasis on minimizing disk I/O cost, meanwhile for smaller databases the emphasis is on minimizing CPU cost. For distributed databases the network cost is also important to consider. We tend to only use one single factor, the disk I/O cost, as it is difficult to assign suitable weights to each cost component.

9.6. Information cost functions

To estimate the cost of different execution strategies, we need to keep track of information for the cost functions. This is stored in the database catalog. The following information can be tracked:

- r_R: Number of tuples for R.
- b_R: Number of blocks occupied in storage for R.
- bfr_R: Number of tuples per block for R.
- $NDV(A, R)$: Number of distinct values of A in R.
- sl_A: Selectivity of attribute A for a condition c. $sl_A = \frac{s_A}{r_R}$
- s_A: Selection cardinality of A. $s_A = sl_A * r_R = \frac{r_R}{NDV(A,R)}$
- x_A: Number of index levels for A.
- b_{I1A}: Number of first level blocks of the index of A.
- n_B: Available buffer space in memory.

Some of the information changes rarely (e.g., x_R), while other information changes frequently (e.g, r_R). Thus, the optimizer will use similar but not exact up-to-date numbers for estimating the cost.

A *histogram* is a data structure (e.g., table) maintained by the DMBs that saves the data distribution. We can use them to help with estimating, as without it we would have to assume aspects about the data such as uniformity or independence that are probably not true. We can store the total tuples per attribute in a *bucket*, that describes the range of possible values. Some variations are used:

- An *equi-width* histogram divides the ranges of values into equal subranges.
- An *equi-height* or *equi-depth* histogram divides the values into buckets of similar size.

9.7. Sorting algorithms

While implementing the query, a common algorithm used is sorting as tuples in a data have no order. We use sorting for **ORDER BY**, sort-merge algorithms (e.g., $\bowtie$, $\cap$, $\cup$), and duplicate elimination (e.g, π).

If all the data fits in the DBMS main memory, we can use standard sorting algorithms (e.g., quicksort). While, *external sorting algorithms* are algorithms suitable for large files that can't fit entirely in main memory. Typically, an *external merge sort* strategy is used for external sorting. The data is split into seperate *runs*, and then combines them into merged sorted runs. The DBMS cache has a buffer space, with each buffer being able to store one disk block. The algorithm has two phases, explained in the following subsections.

Sorting phase

The *runs* (portions or pieces) that are available in the buffer are sorted using an internal algorithm. Then, the results are written back as temporary sorted runs.

Merging phase

The runs are merged with *merge pases*. The *degree of merging* (d_M) are the sorted subfiles that can be merged in each step with at least one buffer block used to save the result. $d_M = min(n_B - 1, n_R)$.

Cost

	Operators	Formula	Example
SOR	n_B = available buffer space b = number of file blocks n_R = number of initial runs	$n_R = \lceil b/n_B \rceil$	$b = 1024$ and $n_B = 5$ disk blocks $n_R = \lceil 1024/5 \rceil = 205$ runs with blocks of 5 except for the last run with 4 blocks
MER	d_M = degree of merging m_P = merge pases	$m_P = \lceil log_{d_M}(n_R) \rceil$	$n_B = 5$, then $d_M = 4$ $m_P = \lceil log_4(205) \rceil = 4$ pases, 4 at a time

The performance of the algorithm can be measured in disk block accesses (either reads or writes) required. The following formula estimates the cost:

$$(2 * b) + (2 * b * (log_{d_M}(n_R))) = 2 * b * (log_{d_M}(n_R) + 1)$$

- $(2 * b)$ represents the number of blocks sorted requiring a read and write from and to memory.
- $(2 * b * (log_{d_M}(n_R)))$ represents the number of block accesses while merging. As each block is merged $(log_{d_M}(n_R)$ passes.
- The minimum requirements are $n_B = 3$, $d_M = 2$ and $n_R = \lceil b/3 \rceil$. The worst-case performance of the algorithm is $(2 \cdot b) + (2 * b * (log_2(\lceil b/3 \rceil)))$.

9.8. Algorithms SELECT operation

The SELECT operation has several algorithms, depending on the type of selection, to search a record in a disk file that satisfies a condition. *Files scans* search file records to find the tuples that satisfy a condition. *Index scans* search indexes to find the tuples that satisfy a condition.

- The optimizer should select the method that *retrieves the fewest records*.
- The *selectivity (sl)* is a ratio (results is between 0 and 1) that is the number of tuples that satisfy a condition to the number of tuples in the relation. 0 represents no records satisfying the condition, while a 1 represents all records.
- *Simple conditions* have only one condition. The methods used to implement the condition are $S1$ through $S7b$.
- *Conjunctive conditions* have multiple conditions separated by **AND**s. We first try methods $S8$, $S9$ or $S10$ for every condition. We can also use $S7a$ or $S7b$. If there are still remaining conditions, we will use $S1$. The form is the following.

$$\sigma_{c_1 AND \cdots AND c_n}(R)$$

- *Disjunctive conditions* have multiple conditions separated by **OR**s. If there is an access path for all the conditions we can use it, but if not we have to use $S1$. We can use any method from $S1$ to $S7$ for every simple condition. We have to remove duplicates. The form is the following.

$$\sigma_{c_1 OR \cdots OR c_n}(R)$$

In the following subsections, each SELECT operation algorithm with their cost is explained.

S1 - Linear search (brute force)

For every file record we test if the condition c is satisfied. Each block is loaded into the main memory to read. We can always use linear search, regardless of how the file is stored. The other algorithms are not always applicable, but are faster than linear search.

The cost for the worst case (e.g., record end key-attribute, non-key attribute selection or no record) we have to check all the b_R.

Cost: b_R

S2 - Binary search

If we are selecting with an equality comparison on an *ordered* file, we can binary search.

For binary search, we can find a key attribute in $log_2(b_R)$ time. Furthermore, if it is a non-key attribute then we might have to retrieve $\lceil(\frac{s_A}{bfr_R})\rceil - 1$. Given that s_A records may satisfy the condition, with $\frac{s_A}{bfr_R}$ file blocks containing the selected records minus the first record retrieved.

Cost: $log_2(b_R) + \lceil(\frac{s_A}{bfr_R})\rceil - 1$

S3a - Primary index

If we are selecting with an equality comparison on a *key attribute* with a primary index, we can use the index to find the unique record.

For every level (x_R) we have to check one disk block to get the results. Plus, one block to access the data from the file.

Cost: $x_R + 1$

S3b - Hash key

If we are selecting with an equality comparison on a *key attribute* with a hash key, we can use the key to find the unique record.

Only one disk block must be accessed to get the result in a normal hash.

Cost: 1

S4 - Multiple primary index

If we are selecting with a comparison condition $\{<, \leq, >, \geq\}$ on a *key attribute* with a primary index, we can find the records using the corresponding equality condition. Then, we can find the corresponding preceding or following records in the ordered file.

For every level (x_R) we have to check one disk block to get the results. Then, half of the file records (b_R) will satisfy the condition c that will be accessed.

Cost: $x_R + \frac{b_R}{2}$

S5 - Multiple clustering index

If we are selecting with an equality comparison on a *non-key attribute* with a clustering index, we can use the index to find all the records.

For every level (x_R) we have to check one disk block to get the results. Given that s_A records may satisfy the condition, with $\frac{s_A}{bfr_R}$ file blocks containing the selected records.

Cost: $x_R + \lceil(\frac{s_A}{bfr_R})\rceil$

S6 - B^+-tree index

If we are selecting a with a secondary B^+-tree index on an *equality comparison*, we can use the tree to find key or non-key values. We can also use a range of values for a range query.

For every level of the tree (x_R) we must check one disk block of the results, plus the reference to the disk block pointer. If the index is on a non-key attribute, s records will satisfy the condition c. If it is a range, it is $x_R + \frac{b_{I1A}}{2} + \frac{r}{2}$.

Cost: $x_R + 1 + s$

S7a - Bitmap index

If we are selecting with a condition based on *values of an attribute*, we can use the corresponding bitmaps with OR or AND operators to get the records.

S7b - Functional key

If the condition involves an *expression* with a functional index, we can use the index to retrieve the records.

S8 - Conjunctive individual index

If an attribute with a single simple condition satisfies the requirements for $S2$, $S3a$, $S3b$, $S4$, $S5$ or $S6$, apply the method and for the remaining simple conditions apply the remaining conjunctive select condition.

S9 - Conjunctive composite index

If we are selecting multiple attributes involved in an equality condition with a composite index or hash (a data structure with both fields combined), we use the additional data structure.

S10 - Conjunctive intersection record pointers

If we are selecting secondary indexes on more than one of the simple conditions with indexes of the *record pointers*, we can get the set of record points and intersect the pointers to gather the records.

The cost estimates for block transfers of selection algorithms $S1$ to $S6$ are shown in Table **??**. For the other algorithms, we can estimate the cost using these methods, depending on the one used.

9.9. Algorithms JOIN operation

The JOIN operation has several algorithms, but it is very time consuming. The following algorithms will provide an overview for EQUIJOIN (or NATURAL JOIN). *Two-way joins* are joins between two files, while if there are more files they are *multi-way joins*. We will only focus on two-way joins.

- The *join selectivity* (js) represents the number of tuples generated after the join. The following formula represents the js of joining the tables R and S with the resulting relationship saved in $T = R \bowtie_c S$.

$$js = \frac{r_T}{r_R * r_S} = \frac{1}{max(NDV(A,R), NDV(B,S))}$$

- A $js = 1$ is the same as applying $\times$.
- The *join cardinality* (jc) is the size of the resulting file after the join operation. The following formula represents the jc.

$$jc = js * r_R * r_S$$

- The results of writing the resulting cost file to disk is represented by the following formula.

$$\frac{js * r_R * r_S}{bfr_R S} = \frac{jc}{bfr_R S}$$

- The *join selection factor* is the fraction of records joined of the relationships.

In the following subsections, each JOIN operation algorithm with their cost is explained.

J1 - Nested-loop join (nested-block join)

For every record r in R, get all records s in S. Then, test $r[A] = s[B]$ for the attributes A of R and B of S. R is the *outer relation* in the *outer loop*, while S is the *inner relation* in the *inner loop*. We then gather the combined records that satisfy the condition.

First, we need to load the times of the outer loop b_R. Then, we need to load every block to memory at least once if they do not fit in memory thus we will perform $b_R * b_S$ accesses. With*more than three buffers* we can perform the operation more efficiently using buffers. It is advantageous to use the relation with fewer blocks as the outer-loop relation due to the effect of the outer-loop relation. **Cost:** $b_R + (\lceil \frac{b_R}{n_B - 2} \rceil * b_S)$

Algorithm:

Example:

For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

Suppose $b_S = 13$, $b_E = 2000$, and $n_B = 3$.

foreach block B_R in R:
    foreach block B_S in s:
        foreach tuple r in B_r:
            foreach tuple s in B_s:
                if ($s == r$):
                    save(s, r)

If we use $ENROLLED$ as the outer relation, then:

$$b_E + (b_E * b_S) = 2000 + (2000 * 13) = 28000$$

If we use $STUDENT$ as the outer relation, then:

$$b_S + (b_S * b_E) = 13 + (13 * 2000) = 26013$$

If we had $n_B = 10$ with $STUDENT$ as the outer relation, then:

$$b_S + (\lceil \frac{b_S}{n_B - 2} \rceil * b_E) = 13 + (\lceil \frac{13}{10 - 2} \rceil * 2000) = 13 + (\lceil \frac{13}{8} \rceil * 2000) = 13 + (2 * 2000) = 4013$$

J2 - Indexed based nested-loop join

If an index or hash key exists for A or B, then the attribute without the structure will be used as the outer loop while the inner structure with the access structure will be the inner loop. We then evaluate the records, gathering those that satisfy the condition.

If there is an index x_B for attribute B of relation S, we can retrieve every record in R and use the index for S. Using the smaller relation with the highest join factor should be more efficient. The cost depends on the type of index.

Cost:

- **Secondary index:** s_B is the selection for the join attribute of B in S.

$$b_R + *(r_R * (x_B + 1 + s_B))$$

- **Clustering index:** s_B is the selection cardinality of B.

$$b_R + *(r_R * (x_B + \frac{s_B}{bfr_B}))$$

- **Primary index:**

$$b_R + *(r_R * (x_B + 1))$$

- **Hask key:** h is the number of blocks estimated to retrieve a record. In linear and static hashing it is estimated as 1, while for extended hashing it is 2.

$$b_R + *(r_R * h))$$

Algorithm:

Example:

For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

foreach tuple r in R:
    foreach tuple s in **Index (**$r_i = s_j$**)**:
        if ($s == r$):
            save (s, r)

Suppose $STUDENT$ has a primary index with $X_{S.id} = 1$. Furthermore, $ENROLLED$ has a secondary index with $X_{E.id} = 2$ and $S_{E.id} = 80$. Also, $b_S = 13$, $b_E = 2000$, $r_E = 10000$, $r_S = 125$, and $n_B = 3$.

If we use $ENROLLED$ as the outer relation, then:

$$b_E + (r_E * (X_{S.id} + 1)) = 2000 + *(10000 * (1 + 1)) = 220000$$

If we use $STUDENT$ as the outer relation, then:

$$b_S + (r_S * (x_{E.id} + 1 + s_{E.id})) = 13 + (125 * (2 + 1 + 8)) = 10388$$

J3 - Sort-merge join

If both R and S are physically sorted by the attributes, we can sort both of the attributes A and B and iterate by joining the smallest of the two values.

If we need to sort each table, we must include this in the cost. If not, the only cost is the merging as we only need to scan once every table to compare the results.

Cost:

- **Sort:** This might not be needed. We use as an example the external merge sort strategy.
$$(2 * b) + (2 * b * (log_{d_M}(n_R))) = 2 * b * (log_{d_M}(n_R) + 1)$$
- **Merge:**
$$b_R + b_S$$
- **Total:**
$$Sort + Merge$$

Algorithm:

//Optional sorts
$cursor_R$ =R.sort(A).cursor()
$cursor_S$ =S.sort(B).cursor()
while ($cursor_R$ AND $cursor_S$): //We stop if there are no more values
    if ($cursor_R > cursor_S$):
        $cursor_S$.next()
    elif ($cursor_R < cursor_S$):
        $cursor_R$.next()
    else:
        save($cursor_R$, $cursor_S$)
        $cursor_S$.next()

Example:
For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

Suppose $b_S = 13$, $b_E = 2000$, $r_E = 10000$, $r_S = 125$, and $n_B = 3$.

If the relations $STUDENT$ and $ENROLLED$ are sorted at the join attribute, then:

$$b_E + b_S = 2000 + 13 = 2013$$

If not then we must sort $STUDENT$ and $ENROLLED$ at the join attribute, thus with external merge sort:

$$dM_S = min(n_B - 1, b_S) = min(2, 13) = 2$$

$$Sort_S = 2 * b_S * (\lceil log_{dM_S}(\lceil \frac{b_S}{n_B} \rceil + 1) \rceil = 2 * 13 * (\lceil log_2(\lceil \frac{13}{3} \rceil)) \rceil + 1) = 26 * (\lceil log_2(5) \rceil + 1) = 104$$

$$dM_E = min(n_B - 1, b_E) = min(2, 2000) = 2$$

$$Sort_E = 2 * b_E * (\lceil log_{dM_E}(\lceil \frac{b_E}{n_B} \rceil) \rceil + 1) = 2 * 2000 * (\lceil log_2(\lceil \frac{2000}{3} \rceil)) \rceil + 1) = 4000 * (\lceil log_2(667) \rceil + 1) = 44000$$

$$b_E + b_S + Sort_E + Sort_S = 2000 + 13 + 104 + 44000 = 46117$$

J4 - Partition-hash join (hash join)

We first start with the *partitioning phase* by saving the partition of each file using a hash function h. We partition both relations. The hash key is the attribute used to join the relations. The records with the same $h(Z)$ value are saved in the same bucket. In the second *probing phase* we scan the other relation to and use the hash function for every join attribute to find an entry in the table with the matching tuple.

To partition both relations, we will require $2(b_R + b_S)$ accessed. We are required to read the blocks and then write them back hashed. Then, in the probing phase we must read $b_R + b_S$ blocks once more.

Cost: $3 * (b_R + b_S)$

Algorithm:

build hash_table HT_R for R
foreach tuple s in S
  if ($h(s)$ in HT_R):
    save (s, r)

Example:
For the query:

$$STUDENT \bowtie_{S.id=E.id} ENROLLED$$

Suppose $b_S = 13$, $b_E = 2000$ and $n_B = 3$. Thus:

$$3 * (b_R + b_S) = 3 * (2000 + 13) = 3 * 2013 = 6039$$

9.10. Review

Admisión UCR

Para que uno pueda entrar en una carrera en la Universidad de Costa Rica (UCR) existe un proceso de admisión. Para estudiantes de nuevo ingreso, se debe tomar un examen de admisión que es administrado anualmente. Luego, las personas que cumplan una nota minima pueden concursar para empadronarse a una carrera. El sistema de concurso de carrera tiene el modelo relacional de la Fig. 9.10.

Carrera:

Código	Nombre	Facultad	Escuela	Recinto

Aplicante:

Cédula	Nombre	Correo	NombreColegio	NotaColegio	NotaAdmisión

Concurso:

CédulaAplicante	CódigoCarrera	Año	Admitido

Figura 9.10: Parte del relacional del sistema de Admisión UCR

Álgebra relacional

Escriba los siguientes queries utilizando los operadores de álgebra relacional. Además, se recomienda escribir las consultas en SQL.

1. Obtenga los nombres de las personas aplicantes.
2. Obtenga el código y nombre de las carreras de la universidad.
3. Obtenga las carreras que son de la facultad de ingeniería.
4. Obtenga los nombres de aplicantes con una nota de admisión mayor que 600 y una nota de colegio mayor que 9.
5. Obtenga todos los concursos a la carrera con código CI (computación) del año 2021.
6. Obtenga todas las personas estudiantes que aplicaron a la carrera de computación en el 2021 que no fueron admitidos.
7. Obtenga el nombre de las personas estudiantes que fueron admitidos en el 2021 a una carrera de la facultad de ingeniería.
8. Obtenga los nombres de las personas que aplicaron en el 2020 **o** en el 2021 a la carrera de computación.
9. Obtenga los nombres de las personas que aplicaron en el 2020 **y** en el 2021 con una nota de admisión mayor a 720.
10. Obtenga el nombre de todas las carreras donde ningún aplicante aplicó en el 2021.

Heurísticas

Para las siguientes expresiones, vamos a usar A para la relación $APLICANTE$, CO para la relación $CONCURSO$ y CA para la relación $CARRERA$. Se resumen algunos atributos por efectos de espacio.

1. Represente los siguientes queries en el *query tree* inicial.
   (a) $\pi_{Escuela}(CA)$
   (b) $\pi_{Cedula,Nombre}(A)$
   (c) $\pi_{Escuela}(\sigma_{Facultad='Ingeneria'}(CA))$
   (d) $\sigma_{CA.Co=CO.CoC}(CA \times CO)$
   (e) $\pi_{NombreColegio}(A \bowtie_{Cedula=CedulaAplicante} CO)$

2. Para los siguientes queries indique cúal es el resultado de aplicar cada paso de las heurísticas.

   (a) Para la expresión $\sigma_{NotaAdmision>420 AND NombreColegio=\text{'High School Musical'}}(A)$ el resultado de **solamente** romper σ es:

   ◯ $\pi_{NotaAdmision>420 AND NombreColegio=\text{'High School Musical'}}(A)$
   ◯ $\sigma_{NombreColegio=\text{'High School Musical'} AND NotaAdmision>420}(A)$
   ◯ $\sigma_{NotaAdmision>420}(\sigma_{NombreColegio=\text{'High School Musical'}}(A))$
   ◯ $\sigma_{NotaAdmision>420 AND NombreColegio=\text{'High School Musical'}}(A)$

   (b) Para la expresión $\sigma_{NotaAdmision>533}(\sigma_{Cedula=CedulaAplicante}(A \times CO))$ el resultado de **solamente** mover σ para abajo es:

   ◯ $((\sigma_{NotaAdmision>533}(A)) \bowtie_{Cedula=CedulaAplicante} CO)$
   ◯ $\sigma_{Cedula=CedulaAplicante}(A \times (\sigma_{NotaAdmision>533}(CO)))$
   ◯ $\sigma_{NotaAdmision>533}((\sigma_{Cedula=CedulaAplicante}(A)) \times CO)$
   ◯ $\sigma_{Cedula=CedulaAplicante}((\sigma_{NotaAdmision>533}(A)) \times CO)$

   (c) Para la expresión $(CA \bowtie_{CA.Co=CO.CoC} (\sigma_{CO.Admitido='Y'}(CO)) \bowtie_{A.Ce=CO.CeA} A)$ el resultado de **solamente** mover σ restrictivo primero es:

   ◯ $(CA \bowtie_{CA.Co=CO.CoC} (\sigma_{CO.Admitido='Y'}(CO)) \bowtie_{A.Ce=CO.CeA} A)$
   ◯ $(A \bowtie_{A.Ce=CO.CeA} (\sigma_{CO.Admitido='Y'}(CO)) \bowtie_{CA.Co=CO.CoC} CA)$
   ◯ $(A \bowtie_{A.Ce=CO.CeA} ((\sigma_{CO.Admitido='Y'}(CO)) \bowtie_{CA.Co=CO.CoC} CA))$
   ◯ $((\sigma_{CO.Admitido='Y'}(CO)) \bowtie_{CA.Co=CO.CoC} CA \bowtie_{A.Ce=CO.CeA} A)$

   (d) Para la expresión $\sigma_{NotaAdmision>533}(\sigma_{Cedula=CedulaAplicante}(A \times CO))$ el resultado de **solamente** convertir $\bowtie$ es:

   ◯ $\sigma_{NotaAdmision>533}(\sigma_{Cedula=CedulaAplicante}(A \bowtie CO))$
   ◯ $\sigma_{NotaAdmision>533}(A \bowtie_{Cedula=CedulaAplicante} CO)$
   ◯ $\sigma_{Cedula=CedulaAplicante}(A \bowtie_{NotaAdmision>533} CO)$
   ◯ $((\sigma_{NotaAdmision>533}(A)) \bowtie_{Cedula=CedulaAplicante} CO)$

   (e) Para la expresión $\pi_{A.Correo}(CO \bowtie_{A.Ce=CO.CeA} A)$ el resultado de **solamente** mover π para abajo es:

   ◯ $(\pi_{A.Ce}(A)) \bowtie_{A.Ce=CO.CeA} (\pi_{A.Correo,CO.CeA}(A))$
   ◯ $\pi_{A.Correo}((\pi_{A.Ce}(A)) \bowtie_{A.Ce=CO.CeA} (\pi_{CO.Correo,CO.CeA}(A)))$
   ◯ $\pi_{A.Correo}(CO \bowtie_{A.Ce=CO.CeA} (\pi_{A.Correo}(A)))$
   ◯ $CO \bowtie_{A.Ce=CO.CeA} (\pi_{A.Correo}(A))$

3. Optimize en su **totalidad** los siguientes queries.

   (a) $\pi_{NombreColegio}(\sigma_{NotaAdmision>750}(A))$

   (b) $\pi_{NombreColegio}(A \bowtie_{Cedula=CedulaAplicante} CO)$

   (c) $\pi_{A.Nombre}(\sigma_{C.Rec=\text{'RFB'} AND A.Ce=CO.CeA AND CA.Co=CO.CoC AND CO.Adm='Y'}(CO \times A \times CA))$

Sorting

Existe una base de datos con tres millones de bloques ($b = 3,000,000$) que se quiere ordenar usando *external merge sort*. n_B son el número de buffers.

1. Si se tienen cuatro buffers, ¿cuántas corridas se requieren para ordenar los bloques?

   ◯ 10 ◯ 15 ◯ 120 ◯ 500,000 ◯ 750,000 ◯ 1,000,000

2. Si se tienen seis buffers, ¿cuántos bloques se pueden ordenar en cada paso del merge?

   ◯ 5 ◯ 6 ◯ 100 ◯ 500,000 ◯ 750,000

3. ¿Cuántos son los buffers mínimos requeridos para poder ordenar con external merge sort?

   ◯ 1 ◯ 2 ◯ 3 ◯ 5 ◯ 10

4. Si se tienen diez buffers, ¿cuántas pasadas se deben realizar para ordenar el archivo?

   ◯ 2 ◯ 4 ◯ 6 ◯ 10 ◯ 100

5. Si se tienen diez buffers, ¿cuánto es el costo en lecturas y escrituras de disco de ordenar?

   ◯ 1,000,000 ◯ 6,000,000 ◯ 18,000,000 ◯ 42,000,000 ◯ 69,000,000

6. ¿Cuánto es el costo en lecturas y escrituras de disco de ordenar con los mínimos requerimientos de merge sort?

   ◯ 31,000,000 ◯ 42,000,000 ◯ 69,000,000 ◯ 120,000,000 ◯ 1,000,000,000

7. ¿Cuál es el número más pequeño de buffers que se pueden tener para que se pueda ordenar en solo dos corridas?

   ◯ 555 ◯ 556 ◯ 1,732 ◯ 1,733 ◯ 1,734 ◯ 2,007 ◯ 2,008 ◯ 2,009

8. Si se tienen veinte buffers, ¿cuál es el número de bloques más grandes que pueden ser ordenados en cuatro corridas?

   ◯ 71 ◯ 556 ◯ 1,001 ◯ 3,702 ◯ 3,703 ◯ 50,007 ◯ 137,180

Join

Suponga que se está realizando un join entre la relación aplicantes (A) y la relación concurso (CO) sobre el atributo $Cedula$ y $CedulaAplicante$. Calcule los costos de lectura y escritura a disco. No incluya los costos de escribir en memoría los resultados.

- Se tienen $n_B = 40$ bloques en el buffer.
- Se ocupan $b_A = 1,300$ bloques para guardar la relación A.
- Se ocupan $b_{CO} = 3,900$ bloques para guardar la relación CO.

1. ¿Cuál es el costo de realizar un *nested loop join* con A como la relación exterior y CO como la relación interior?

   ◯ 124,000 ◯ 158,000 ◯ 137,800 ◯ 210,000 ◯ 269,900

2. ¿Cuál es el costo de realizar un *nested loop join* con CO como la relación exterior y A como la relación interior?

   ◯ 124,000 ◯ 158,000 ◯ 137,800 ◯ 210,000 ◯ 269,900

3. ¿Cuál es el costo de particionar en *partition-hash join*?

   ◯ 2,080 ◯ 4,160 ◯ 6,240 ◯ 8,320 ◯ 10,400

4. ¿Cuál es el costo de sondear en *partition-hash join*?

   ◯ 1,900 ◯ 3,000 ◯ 4,100 ◯ 5,200 ◯ 6,300

5. ¿Cuál es el costo total de *partition-hash join*?

   ◯ $3,980$ ◯ $4,160$ ◯ $7,160$ ◯ $10,340$ ◯ $15,600$ ◯ $16,700$

6. ¿Cuál es el costo de ordenar A para el atributo $Cedula$?

   ◯ $5,200$ ◯ $6,400$ ◯ $7,600$ ◯ $8,800$ ◯ $10,000$

7. ¿Cuál es el costo de ordenar CO para el atributo $CedulaAplicante$?

   ◯ $5,400$ ◯ $10,600$ ◯ $20,100$ ◯ $23,400$ ◯ $25,600$

8. ¿Cuál es el costo de realización del merge en *sort-merge* join entre A y CO si no hay duplicados?

   ◯ $1,300$ ◯ $3,900$ ◯ $5,200$ ◯ $10,400$ ◯ $14,400$

9. En el peor caso, ¿cuál es el costo de *sort-merge* join?

   ◯ $50,720,000$ ◯ $72,600,000$ ◯ $83,000,200$ ◯ $111,200,000$ ◯ $121,680,000$

10. ¿Cuál es el costo total de *sort-merge* join entre A y CO si no hay duplicados?

    ◯ $5,200$ ◯ $28,600$ ◯ $30,120$ ◯ $33,800$ ◯ $38,600$

11. ¿Cuál es el algoritmo más eficiente? Explique.

    ◯ *Sort-merge join* ◯ *Nested loop join* con A como relación exterior. ◯ *Nested loop join* con CO como relación exterior. ◯ *Partition-hash join*.

Steam

Steam es uno de los distribuidores de juegos más grande del mundo. La compañía ha distribuido más de $50,000$ juegos de PC en centenas de géneros. Dado la calidad de su plataforma, actualmente cuenta con más de $1,000,000,000$ (miles de millones) de cuentas. Cada cuenta puede comprar diversos juegos de los cuales la plataforma recolecta información de uso. La base de datos de Steam tiene el modelo relacional de la Fig. 9.11 para las tablas que se relacionan con la distribución de uso.

JUEGO (J):

ID	Nombre	Descripcion	FechaLanzamiento	Genero

CUENTA (CU):

ID	Username	Correo	Nivel	Pais	Saldo

COMPRA (CO):

Juego	Cuenta	MinutosJugados	FechaCompra	UltimaFecha

Figura 9.11: Parte del relacional del sistema de Steam

1. Escriba la expresión de álgebra relacional que obtenga todos los géneros de juegos que se han jugado por más de 50 horas ($3,000$ minutos).

2. Se tiene la siguiente expresión de álgebra relacional:

   $$\pi_{J.Nombre}(\sigma_{J.ID=CO.Juego\ AND\ CU.ID=CO.Cuenta\ AND\ CU.Pais='CR'\ AND\ CO.UF>='01-01-2021'}(CU \times CO \times J))$$

   Nota: $CO.UF = CO.UltimaFecha$

(a) Explique brevemente, en sus propias palabras, qué consulta realiza la expresión de álgebra relacional.

(b) Dibuje el árbol canónico de la expresión de álgebra relacional.

(c) Optimice en totalidad, utilizando heurísticas, la expresión de álgebra relacional. Debe escribir el resultado de cada paso en un árbol de consulta (*query tree*).

3. Suponga que se está realizando un join entre CUENTA (CU) y COMPRA (CO) sobre los atributos ID y $Cuenta$, respectivamente. Estime el costo de lectura y escritura a disco. No debe incluir los costos de escribir en memoría los resultados.

   - Se tienen $n_B = 100$ bloques en el buffer.
   - Se ocupan $b_{CU} = 5,500$ bloques para guardar CU.
   - Se ocupan $b_{CO} = 10,200$ bloques para guardar CO.

   (a) ¿Cuál es el costo de realizar un *nested loop join* con CU como relación exterior y CO como la relación interior?

   (b) ¿Cuál es el costo de realizar un *nested loop join* con CO como relación exterior y CU como la relación interior?

   (c) ¿Cuál es el costo de realizar un *sort-merge join* entre CO y CU si no hay duplicados? Las relaciones **NO** se encuentran ordenadas por los atributos ID y $Cuenta$ en memoría.

   (d) ¿Cuál es el costo de realizar un *partition-hash join* con CU y CO?

   (e) Para estas estimaciones, ¿cuál es el algoritmo más eficiente? Explique.

Parte VI

Transacciones

Capítulo 10

Transacciones

10.1. Introduction

It is common that when we execute operations in a database, from the point of view of a user it seems as a *single logical work unit*. However, frequently we require multiple operations to perform a work unit, requiring either *all* the work to be executed or *none at all* if there is a failure. Per example, if Alice wants to transfer $\$100$ to Bob's bank account (logical work unit), the transfer requires to:

1. Check if Alice has enough funds ($\$100$) to transfer to Bob.
2. Debit the $\$100$ out of Alice's bank account.
3. Add the $\$100$ to Bob's bank account.

Other examples of systems that use transactions include ticket reservation, and retail purchasing. To manage these types of operations, we need to perform the collection of operations as one logical work unit called a *transaction*.

10.2. Definition

A database operates over data items $A, B, C, \cdots$. A *data item* can be an attribute, tuple, page, disk block, table or database. Furthermore, we can perform the following operations:

- *read(X):* Reads the data item X from the database.
- *write(X):* Writes the variable X to a data item of the database. We asume that the writes are saving directly to disk, though that might is usually not the case as it is probably saved in cache or a buffer.

A *transaction* can be represented, in a simplified form, as a collection of $read(X)$ and $write(X)$. A transaction is delimited between a *begin transaction* and *end transaction*. A transaction has the following elements:

- **BEGIN:** Where the transaction strats. In SQL, it is represented by a **BEGIN**.
- **READ or WRITE:** $read(X)$ or $write(X)$ operations. $read(X)$ can also be represented as $R(X)$, while $write(X)$ as $W(X)$.
- **COMMIT:** A successful end to the transaction that *commits* the updates safely to the database. In SQL, it is represented by a **COMMIT**.

- **ROLLABACK:** An unsuccessful end to the transaction, requiring to *undo* the database changes. In SQL, it is represented by an **ABORT**.

The states of a transaction are shown in Fig. 10.1. A transaction starts in the *active state* after beginning and while executing $read(X)$ or $write(X)$. When the transaction has reached the end, it goes to the *partially committed* state. If there are no errors, the transaction ends as a *committed* transaction. If there is a failure while executing operations or committing the results, the transaction enters a *fail state*. Thus the results to the database are then *aborted*. After being aborted or commited, the transaction is *terminated*.

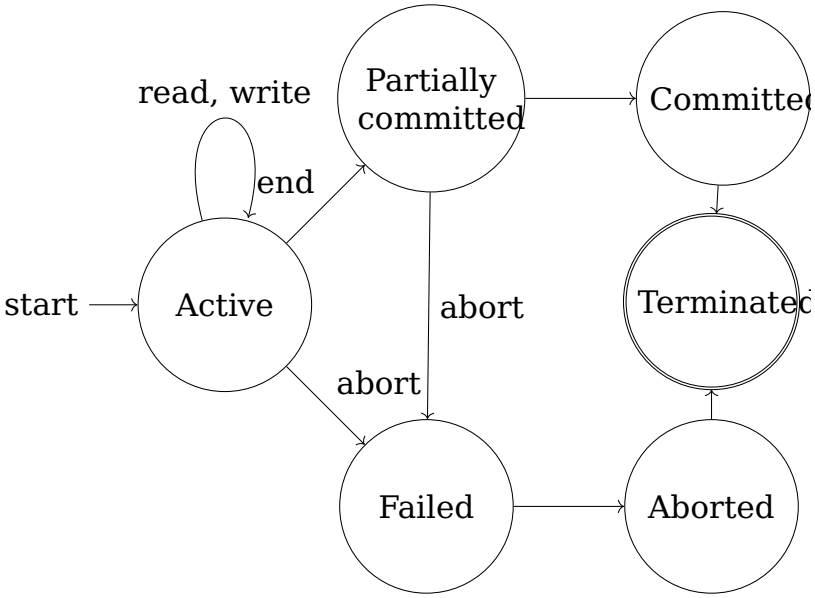


Figura 10.1: States of a transaction

An example of a transaction T_1 is shown in Fig. 11.2. In this transaction, $\$100$ are transfered from Alice's bank account (A) to Bob's bank account (B). This transaction can also be represented in one line as $R(A), W(A), R(B), W(B)$. Another example is T_2 shown in Fig. 11.3. In T_2, $\$50$ have been deposited to Alice's bank account (A). Sometimes the **BEGIN** and **COMMIT** statements are omitted, though they do exist for a transaction.

T_1 :
BEGIN;
read(A);
$A := A - 100$;
write(A);
read(B);
$B := B + 100$;
write(B);
COMMIT;

Figura 10.2: Transfering $\$100$ from Alice's bank account (A) to Bob's bank account (B)

10.3. Transaction properties

While transactions are executing, we require that *ACID properties* are maintained for the database. We will describe each property with transaction T_1.

T_2 : BEGIN;
read(A);
$A := A + 50$;
write(A);
COMMIT;

Figura 10.3: Cash deposit of $\$50$ to Alice's bank account (A)

Atomicity

Let us assume that account A has $\$1000$ and account B has $\$500$ before the transaction. Suppose that there is a failure after the $write(A)$ operation and before the $write(B)$ operation. Thus, the value would have $\$900$ for A and $\$500$ for B, destroying $\$100$. The database would be in an *inconsistent state* as the system would no longer reflect the real world state. This inconsistent state will happen at one point in the transaction, which will then become consistent after the $write(B)$ operation. Thus, if the transaction was never started or was guaranteed to be completed, the inconsistent state would not be visible except during the transaction.

The *atomicity* means that all operations are executed or none at all. During the transaction the system keeps track of the changes in a *log* file. If there is any failure, the system restores the old values with the log files as if the transaction was not executed. The database system is responsible for ensuring the atomicity through the *recovery system*.

Consistency

We want that the result $A+B$ to be the same *before and after executing the transaction*. Without consistency, money could be destroyed or created (no bueno).

The *consistency* property ensures that executing a transaction by itself preserves the consistency of the database. The *programmer* who codes the transaction is responsible to ensure the consistency.

Isolation

If several transactions are executed concurrently, operations may interleave in such a way that may produce an inconsistent state. If while transferring money from A to B between $write(A)$ and $write(B)$ operation another second transaction read data reads A or B there will be an inconsistent state. If this second transaction updates A and B based on the inconsistent values, the database will be inconsistent after the execution of the transactions.

The *isolation* property ensures that executing the transaction concurrently results in an equivalent state as if it was done one at a time in a particular order. The database ensures this property with the *concurrency-control system*.

Durability

If we have transferred the funds successfully, even if there is a system failure later the data cannot be lost. For example, the system cannot lose that A has $\$900$ in the account and B has $\$600$ in the account.

The *durability* property guarantees after successfully completing a transaction, all the updates persist even if there is a system failure. To do this, we must write to

disk before finishing the transaction or by saving enough information to disk to be able to reconstruct the updates after failure. The *recovery system* is also in charge of ensuring durability.

Capítulo 11

Control de la concurrencia

11.1. Introduction

Usually, transaction processing systems allow multiple transactions to run concurrently causing inconsistencies with the data. Though running transaction *serially* (one at a time) is easier, there are benefits for using concurrency:

- **Improved throughput and resource utilization.** As there are many steps in a transaction, some involving the CPU and others the I/O, both can be executed in parallel. One can have one transaction reading from a disk and another using the CPU. If there are multiple disks, another transaction can also read from it. Therefore the number of transactions executed at a time (*throughput*), and idle time for CPU and disks reduce (*resource utilization*).

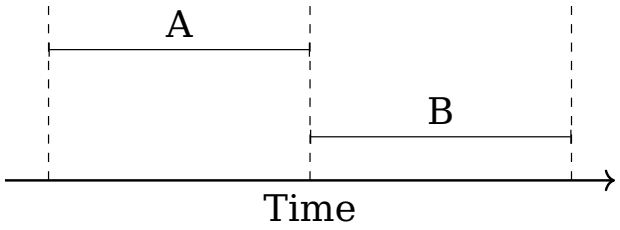
- **Reduced waiting time.** The running time of a transaction may vary, thus if transactions run serially a short transition might have unpredictable delays waiting for a longer transaction to finish. Running a transaction concurrently, if they use different parts for the database, is better as it allows them to share resources. Thus it reduces unpredictable delays, bringing down the average time that a transaction takes to be completed after submission (*average response time*).

Still, as concurrency allows for multiple transactions to be run at the same time we must ensure the *isolation* ACID property. To do this, we must use *schedules* to identify the order of execution (Sections 11.3). Furthermore to ensure the consistency of the database we use several mechanisms called *concurrency-control protocols* (Section 11.6).

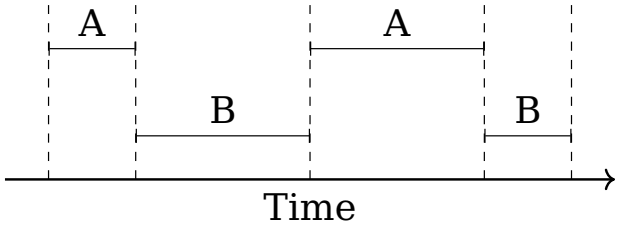
11.2. Concurrency

While some systems may have a database system with a single-user, it is common for database systems to be *multi-user*. A simple way we could handle multiple transactions is executing one after the other with only one thread (Subfig. 11.1a). However, this is inefficient in time as there is only one transaction executing and we require to copy the database per transaction.

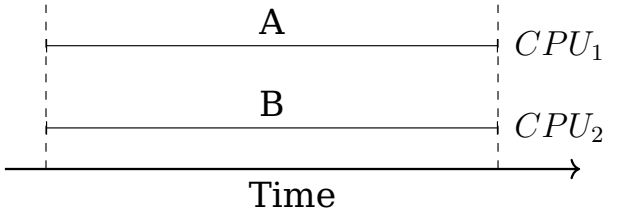
Another possible approach is to process the transaction *concurrently* in a thread using *interleaved processing* by partially executing the transaction and then suspending it to execute other operations (Subfig. 11.1b). Furthermore, we could also use multiple threads to allow for *parallel processing* (Subfig. 11.1c).



(a) Serial processing



(b) Interleaved processing



(c) Parallel processing

Figura 11.1: Execution of processes A and B

11.3. Schedules

The chronological order of execution of sequences of n transactions $T_1, T_2, \cdots, T_n$ is the *schedule* or *history*. The order of the instructions inside the transaction must be preserved.

Suppose we have a transaction T_1 that transfers $\$100$ from Alice's bank account (A) to Bob's bank account (B) shown in Fig. 11.2. We also have a transaction T_2 in which $\$50$ are deposited in cash to Alice's bank account (A) shown in Fig. 11.3. Therefore a schedule would be the order of execution for both of these transactions. At the end of each transaction to detail that the transaction has entered a committed state, we use the *commit* instruction.

T_1 :
read(A);
$A \coloneqq A - 100$;
write(A);
read(B);
$B \coloneqq B + 100$;
write(B);

Figura 11.2: Transfering $\$100$ from Alice's bank account (A) to Bob's bank account (B)

An example of a *serial* schedule is shown in Fig. 11.4. The instructors are shown in chronological order from top to bottom, with the instructions for T_1 and T_2 shown in the left and right column, respectively. If there was $\$1000$ in A and $\$500$ in B, the result of the scheduled execution is $\$950$ in A and $\$600$ in B. As the total money in the

T_2 :	read(A);
	$A \coloneqq A + 50$;
	write(A);

Figura 11.3: Cash deposit of $\$50$ to Alice's bank account (A)

accounts is equal to $A + B + \$50$, the data is consistent. Furthermore, if T_2 and then T_1 is executed, shown in Fig. 11.5, the result will also be consistent.

T_1	T_2

read(A);
$A \coloneqq A - 100$;
write(A);
read(B);
$B \coloneqq B + 100$;
write(B);
commit;	

	read(A);
	$A \coloneqq A + 50$;
	write(A);
	commit;

Figura 11.4: Schedule 1. Serial schedule of T_1 after T_2

T_1	T_2

	read(A);
	$A \coloneqq A + 50$;
	write(A);
	commit;

read(A);
$A \coloneqq A - 100$;
write(A);
read(B);
$B \coloneqq B + 100$;
write(B);
commit;	

Figura 11.5: Schedule 2. Serial schedule of T_2 after T_1

We could also execute transactions *concurrently*, generating many possible sequences as operations that can be *interleaved*. However, we cannot predict how many instructions will be executed by the CPU before switching to another transaction.

A posible schedule is shown in Fig. 11.6, where the total money in the accounts after the transaction is $A + B + \$50$. Thus this schedule was equivalent to one that was executed serially. However, not all concurrent executions will result in a consistent state. The schedule shown in Fig. 11.7 after execution will result with $\$900$ in A and $\$600$ in B. Therefore, it is an inconsistent state as the $\$50$ deposit is lost. Therefore, we cannot leave to the operating system the control of possible schedules as it may result in inconsistent states. The database system will be in a chart of executing

schedules that result in consistent states using the *concurrency-control* component. Therefore, to ensure the consistency a concurrent schedule must be equivalent to a serial schedule. Such schedules are *serializable* (Section 11.5).

T_1	T_2
read(A);	
$A \coloneqq A - 100$;	
write(A);	
	read(A);
	$A \coloneqq A + 50$;
	write(A);
	commit;
read(B);	
$B \coloneqq B + 100$;	
write(B);	
commit;	

Figura 11.6: Schedule 3. Concurrent schedule equivalent to serial execution

T_1	T_2
read(A);	
$A \coloneqq A - 100$;	
	read(A);
	$A \coloneqq A + 50$;
	write(A);
	commit;
write(A);	
read(B);	
$B \coloneqq B + 100$;	
write(B);	
commit;	

Figura 11.7: Schedule 4. Concurrent schedule resulting in an inconsistent state

11.4. Concurrency-control problems

There are many problems that can happen while we execute our queries concurrently.

Lost update

A *lost update* (dirty write) occurs when two transactions interleave in such a way that the value produced by the database is incorrect. W_i represents a write to transaction T_i and a R_i a read to transaction T_i. The anomaly occurs for T_i when the schedule interleaves for Q in such a way where $W_i(Q), \cdots, W_j(Q)$.

An example is shown in Fig. 11.8. When T_2 writes the result of A it is incorrect as it reads the value of A before T_1 changes the database result. Thus, the result in A will be $\$1050$ instead of $\$950$ due to losing the debit of $\$100$ to the bank account.

T_1	T_2
read(A);	
$A \coloneqq A - 100$;	
	read(A);
	$A \coloneqq A + 50$;
write(A);	
	write(A);
read(B);	
$A \coloneqq B + 100$;	
write(B);	

Figura 11.8: Lost update example

Dirty read

A *dirty read* (temporary update) occurs when a transaction updates a value that then fails, while another transaction reads the data before the roll back. The problem occurs for T_i when the schedule interleaves for Q in such a way where $W_j(Q), \cdots, R_i(Q)$.

An example is shown in Fig. 11.9. T_2 reads the temporary result of A. However, T_1 has a failure and the previous data is recovered to the original value. Therefore, T_2 reads *dirty data* created by the transaction that has not been completed and committed.

T_1	T_2
read(A);	
$A \coloneqq A - 100$;	
write(A);	
	read(A);
	$A \coloneqq A + 50$;
	write(A);
read(B);	
ABORT;	

Figura 11.9: Dirty read example

Unrepeatable read

An *unrepeatable read* (fuzzy or non-repeatable read) occurs when a transaction reads the same data twice, but the value was changed by another transaction between reads. The issue occurs for T_i when the schedule interleaves for Q in such a way where $R_i(Q), \cdots, W_j(Q), \cdots R_i(Q)$.

An example is shown in Fig. 11.10. T_1 reads a result for A at the beginning of the transaction that is modified by T_2. When T_1 reads the data of A again there are *different values* for the same data.

Phantom read

A phantom read occurs when a transaction repeats a *search condition* but gets a different a *set of items* that satisfies the condition. [y in Q] represents modifying (inserting, updating or deleting) a tuple y for the data item Q . The anomaly occurs

T_1	T_2
read(A**);**	
$A \coloneqq A - 50$;	
write(A);	
	read(A);
	$A \coloneqq A + 50$;
	write(A**);**
read(A**);**	
$A \coloneqq A - 50$;	
write(A);	

Figura 11.10: Unrepeatable read example

for T_i when the schedule interleaves for Q in such a way where $R_i(Q), \cdots, W_j[y$ in $Q], \cdots R_i(Q)$.

An example is shown in Fig. 11.11. T_1 reads the total of tA money transferred in Alice's bank account, However, in T_2 a new transfer is added to all the money transfers tA. Therefore, when the same condition is executed in T_1 a new *phantom tuple* exists that was not previously there.

T_1	T_2
read(tA**);**	
	insert(a_{n+1} **in** tA**);**
read(tA**);**	

Figura 11.11: Phantom read example

11.5. Serializability

A *serializable* schedule determines if the order of concurrent operations are equivalent to the serial execution. The *serializability* identifies when a schedule is serializable. Serial schedules are serializable, and interleaved executions can also be serializable though it is harder to determine. There are two types of serializability: conflict and view. Neither definition encapsulates all serializable schedules.

Conflict serializability

If we have a schedule S with two consecutive instructions I and J of transactions T_i and T_j, with $i \neq j$. If I and J are different data items the steps can be swapped without affecting the results. However if I and J both refer to the same data item Q then the order may matter. As in our transactions we only have *read* and *write* instructions only four possibilities can happen:

- **read-read:** $I = read(Q), J = read(Q)$. The order does not matter as, regardless of the order of I and J, the value Q will be the same.
- **read-write:** $I = read(Q), J = write(Q)$. The order matters. If T_i first reads with instruction I Q and then T_j writes to Q a value, then T_i will not have the value written by T_j. However, if T_j writes first a value in Q in instruction T, then T_i will read in I the updated Q value.

- **write-read:** $I = write(Q), J = read(Q)$. The order does matter, for the same reason as the previous case.
- **write-write:** $I = write(Q), J = write(Q)$. The order does not affect T_i and T_j. However, for the next $read(Q)$ operation the order matters as only the last $write$ instruction is preserved in the database. If there is no other $write(Q)$ the order does affect the final value of Q in the database.

Therefore, if either the I or J instruction performs a $write$ operation on the same data item there is a *conflict*. For example, the schedule 3 shown in Fig. 11.6 has a conflict between the $write(A)$ of T_1 with $read(A)$ of T_2. However, $write(A)$ of T_2 has no conflict with $read(B)$ of T_1 as they access different data items.

If I and J are two consecutive instructions of a schedule S that do not have a conflict, then we can swap the order of I and J to generate a new schedule for S'. S is equivalent to S' as all instructions have the same order except for I and J whose order does not matter. If schedule S can be transformed into a schedule S' by swapping non-conflicting instructions, then S and S' are *conflict equivalent*.

Fig. 11.12 shows the swaps to transform the schedule 3 (Fig. 11.6) to the serial schedule 1 (Fig. 11.4). We only swap the *reads* and *writes* as these are the only operations considered.

T_1	T_2
read(A);	
write(A);	
	read(A);
	write(A);
read(B);	
write(B);	

(a) Concurrent schedule

T_1	T_2
read(A);	
write(A);	
	read(A);
read(B);	
	write(A);
write(B);	

(b) T_1 $read(B) \leftrightarrow T_2$ $write(A)$

T_1	T_2
read(A);	
write(A);	
	read(A);
read(B);	
write(B);	
	write(A);

(c) T_1 $write(B) \leftrightarrow T_2$ $write(A)$

T_1	T_2
read(A);	
write(A);	
read(B);	
	read(A);
write(B);	
	write(A);

(d) T_1 $read(B) \leftrightarrow T_2$ $read(A)$

T_1	T_2
read(A);	
write(A);	
read(B);	
write(B);	
	read(A);
	write(A);

(e) T_1 $read(B) \leftrightarrow T_2$ $read(A)$

Figura 11.12: Transforming a concurrent schedule to an equivalent serial schedule

A schedule S is *conflict serializable* if it is conflict equivalent to a serial schedule. For example, schedule 3 is conflict serializable to schedule 1. However, schedule 4 (Fig. 11.7) is not conflict serializable as it is not equivalent to either schedule 1 or schedule 2.

Another way to determine if the conflict serializability of a schedule is with a *dependency graph* or *precedence graph*. We create a node for each of the following operations. The graph G will have pairs of $G = (V, E)$. The set of vertices V are all the transactions in the schedule. The edges E are created if for T_i to T_j if any of the following conditions (conflicts) hold:

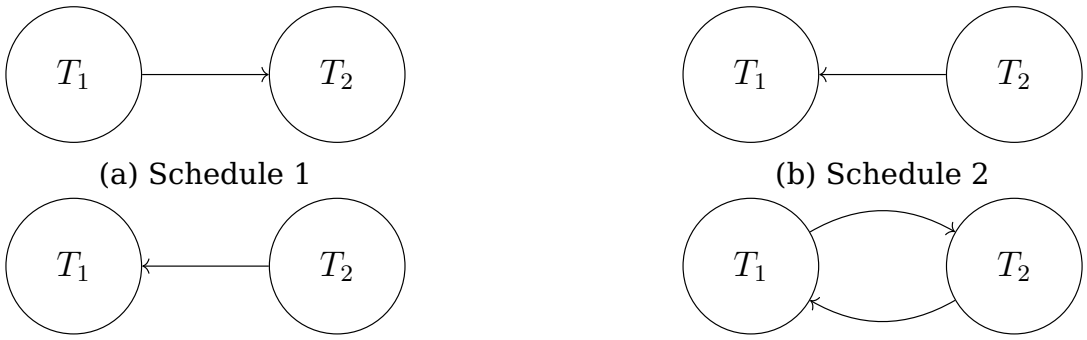
- T_i executes a $write(Q)$ before T_j executes a $read(Q)$.
- T_i executes a $read(Q)$ before T_j executes a $write(Q)$.

- T_i executes a $write(Q)$ before T_j executes a $write(Q)$.

If there is an edge T_i to T_j exists in the graph, then in a serial execution S' T_i must execute before T_j. If there are no cycles, then the schedule S is conflict serializable. If the graph has a cycle, then S is not conflict serializable.

For example, the precedence graphs of schedules 1 to 4 are shown in Fig. 11.13.

- For schedule 1 (Subfig. 11.13a) there is a single edge $T_1 \rightarrow T_2$ as T_1 executes $write(A)$ before T_2 executes a $read(A)$.
- Schedule 2 (Subfig. 11.13b) has also only one edge $T_2 \rightarrow T_1$ for a similar reason than Schedule 1. The edge from $T_2 \rightarrow T_1$ exists as T_2 executes $write(A)$ before T_2 executes a $read(A)$.
- Schedule 3 (Subfig. 11.13c) has one edge $T_1 \rightarrow T_2$ as T_1 executes $write(A)$ before T_2 executes a $read(A)$. Therefore, the concurrent schedule 3 is conflict serializable.
- For schedule 4 (Subfig. 11.13d), there is an edge $T_1 \rightarrow T_2$ as T_1 executes $read(A)$ before T_2 executes a $write(A)$. There is also an edge $T_2 \rightarrow T_1$ as T_2 executes $write(A)$ before T_2 executes $write(A)$. Therefore, there is a cycle and the schedule is not serializable.



(a) Schedule 1 (b) Schedule 2

(c) Schedule 3 (d) Schedule 4

Figura 11.13: Precedence graph of schedules 1 through 4

The *serializability order* defines the order of execution of the transactions that is consistent with the partial order of the precedence graph. To test for conflict serializability, the precedence graphs is constructed and a cycle-detection algorithm used.

View serializability

If there is a transaction T_3, shown in Fig. 11.14, that transfers $\$20$ from Bob's bank account (B) to Alice's bank account (A). A schedule 5 could be created by executing T_1 and T_3, as shown in shown in Fig. 11.15. The precedence graph of schedule 5 has an edge from $T_1 \rightarrow T_3$ as T_1 executes $write(A)$ before T_3 executes $read(A)$, and an edge from $T_3 \rightarrow T_1$ as T_3 executes $write(B)$ before T_1 executes $read(B)$. Thus, there is a cycle and the transaction is not conflict serializable.

However, if we execute the transaction we will get an equivalent result to a serial schedule of $< T_1, T_3 >$ due to the fact that the mathematical increment and decrement operations are commutative. Therefore, there are schedules that produce the same outcome but are not conflict equivalent. Analyzing the results instead of only considering *read* and *write* operations is the *view serializability*. However, this type of serializability is not used in practice because it is computationally complex to determine (NP-complete).

T_3 :	read(B);
	$B \coloneqq B - 20$;
	write(B);
	read(A);
	$A \coloneqq A + 20$;
	write(A);

Figura 11.14: Transfering $\$20$ from Bob's bank account (A) to Alice's bank account (B)

T_1	T_3

read(A);
$A \coloneqq A - 100$;
write(A);	

	read(B);
	$B \coloneqq B - 20$;
	write(B);

read(B);
$B \coloneqq B + 100$;
write(B);	

	read(A);
	$A \coloneqq A + 20$;
	write(A);

Figura 11.15: Schedule 5. Concurrent view serializable schedule of T_1 and T_3

11.6. Concurrency-control protocols

To ensure the *isolation* ACID property, there are several possible mechanisms or techniques known as *concurrency-control protocols* or *concurrency-control schemes*. These protocols ensure the proper execution of transactions when they are concurrent and interleaved. Thus, they generate an execution schedule equivalent to a serial schedule. Furthermore, the protocols cannot know if there are conflicts ahead of time.

There are two main categories for the protocols:

- **Pessimistic:** The DBMS assumes that transactions *will have conflicts*, therefore it doesn't allow the problems to occur. Lock-based (Section 11.7) are pessimistic.
- **Optimistic:** The DBMS assumes that *conflicts between transactions are rare*, therefore it assumes it will be able to finish the transaction after commiting. Checks are performed after the transaction is executed. Validation-based protocols (Section 11.11) and timestamp-based protocols (Section 11.10) are optimistic.

11.7. Lock-based protocols

To ensure that only one transaction is modifying a data item at a time (*mutually exclusive manner*) a *lock* can be used on the data item. The basic types of locks are:

- **Shared lock ($S - LOCK$).** Several transactions can read at the same time, but

none can write. This lock can be acquired by multiple transactions at the same time.

- **Exclusive lock ($X-LOCK$).** Only one transaction can both read and write. This lock prevents other transactions acquiring $S-LOCK$ or $X-LOCK$.

We request the $S-LOCK$ for a data item Q executing the **S-LOCK(Q)** instruction, while for $X-LOCK$ we execute the **X-LOCK(Q)** instruction. To release either lock on a data item Q, we execute the **UNLOCK(Q)** instruction. If we hold the lock till the end of a transaction (after a commit or abort) it is a *long lock*, while if we liberate it before it is a *short lock*.

The transactions *request* locks (or upgrades) to the *concurrency-control manager*. The lock requested depends on the type of transaction performed. The transaction will have to *wait* to continue it's execution until the concurrency-control manager *grants* the lock. The lock will be granted until when all the incompatible locks held by other transactions have been released. When a transaction finishes using a lock, it must be *released* so other transactions can use it.

Compatible modes do not require waiting to be granted permission to use the lock if another transaction currently holds a lock. The compatibility between the modes is shown in Fig. 11.16. Only $S-LOCK$s are compatible with each other and can be held simultaneously.

	$S-LOCK$	$X-LOCK$
$S-LOCK$	✓	✗
$X-LOCK$	✗	✗

Figura 11.16: Compatibility matrix of $S-LOCK$ and $X-LOCK$

Locks can have different types of issues:

- If we release a lock too soon after reading or writing, an *inconsistent state* may occur. For example, schedule 6 shown in Fig. 11.17 there is an inconsistent state for T_1 as it was modified by T_2 during its execution. Therefore, locks by themselves do not ensure serializability. Inconsistent states can cause real-world problems, thus these cannot be tolerated by the DBMS.

T_1	T_2

X-LOCK(A);
read(A);
write(A);
UNLOCK(A);	

	X-LOCK(A); write(A); **UNLOCK(A);**
S-LOCK(A); read(A); **UNLOCK(A);**	

Figura 11.17: Schedule 6. Inconsistent state for transaction T_1

- If a transaction is waiting for a lock that is not granted, the transaction is *starved* and cannot progress. For example, schedule 7 shown in Fig. 11.18. T_1 has a

$S-LOCK$ granted, while T_2 requests the lock but is waiting to get granted access to A. However, if starvation is not considered it will allow for T_3 to be granted the lock. Thus, even if T_1 has unlocked A, T_2 is still waiting. This can further happen with transaction $T_4, \cdots, T_n$. Thus, T_2 cannot progress as it is left waiting by the lock manager for the $X-LOCK$ on A. Starvation can be handled if a lock request is not blocked by a later request.

T_1	T_2	T_3	T_4	$\cdots$
S-LOCK(A);				$\cdots$
	X-LOCK(A);			$\cdots$
		S-LOCK(A);		$\cdots$
UNLOCK(A);				$\cdots$
			S-LOCK(A);	$\cdots$
		UNLOCK(A);		$\cdots$

Figura 11.18: Schedule 7. Starvation of transaction T_2

- If two transactions are waiting for the other to release a lock, a *deadlock* can occur. Fig. 11.19 shows an example of a schedule that generates a deadlock. When T_2 ask for an $S-LOCK$ on A it is not granted as T_1 still holds an $X-LOCK$ on A. The deadlock is then generated when T_1 asks for an $X-LOCK$ on B as it cannot be granted due to T_2 having an $S-LOCK$ on B. Therefore, neither T_1 or T_2 can advance. Deadlocks are a necessary evil as they can be handled by rolling back transactions (Section 11.8).

T_1	T_2
X-LOCK(A);	
	S-LOCK(B);
	read(B);
	S-LOCK(A);
write(A);	
UNLOCK(B);	

Figura 11.19: Schedule 8. Deadlock of data items A and B between two transactions

Locking protocols define a set of rules that must be followed to lock and unlock data items. These protocols restrict the possible schedules and do not produce all possible serializable schedules. In the following subsections, these protocols will be defined. The relationship between these protocols and how long the locks are held is shown in Fig. 11.20.

	$S-LOCK$	$X-LOCK$
2PL	short	short
Strict 2PL	short	long
Rigorous 2PL	long	long

Figura 11.20: Lock liberation of 2PL protocols

Two-phase locking

The *two-phase locking protocol* (2PL) or *basic 2PL* ensures serializability by issuing locks and unlocks in two phases:

1. **Growing or expanding phase.** Transactions start by obtaining locks without releasing them.
2. **Shrinking phase.** After locks are no longer needed, the transaction releases locks and cannot obtain any new ones.

Both $S-LOCK$s and $X-LOCK$s are *short locks*. A schedule example is shown in Fig. 11.21. First, both transactions T_1 and T_2 ask for all the locks for all the data items used at the beginning of the transaction. For T_1, there is an $X-LOCK$ for A and B, while for T_2 it is a $X-LOCK$ for A. After T_1 completely finishes using the $X-LOCK$ on A, it is released thus T_2 can continue executing the operations. Furthermore, as A is not used any more in T_1, there will be no inconsistent data states.

T_1	T_2
X-LOCK(A);	
X-LOCK(B);	
read(A);	
write(A);	
	X-LOCK(A);
UNLOCK(A);	
	read(A);
	write(A);
read(B);	
write(B);	
ABORT;	

Figura 11.21: Schedule 9. 2PL with cascading aborts

However, there are several limitations to 2PL. Deadlocks can still occur and it limits concurrency. Furthermore, *cascading aborts* or *cascading rollbacks* can happen when a transaction aborts and another transaction must be rolled back. For example, in schedule 9 of Fig. 11.21 there is an abort at the end of T_1, thus T_2 must also abort the changes. Thus the effort of executing the instructions in T_2 is wasted.

Strict two-phase locking

To avoid *cascading rollbacks*, we can use a modification of 2PL called *strict two-phase locking protocol*. The $X-LOCK$s are only released until the transaction commits or aborts. Therefore, $S-LOCK$s are *short locks* and $X-LOCK$s are *long locks*. For example, Fig. 11.22 shows schedule 10 where all the locks are requested at the beginning of the execution of T_1 and T_2. T_1 can release the $S-LOCK$ on B before the end of the transaction, however the $X-LOCK$ on A is released when the transaction will be committed. Until then can the $X-LOCK$ on A be granted for T_2 eliminating the cascading rollbacks.

Rigorous two-phase locking

Another variant is called *rigorous or strong strict two-phase locking protocol* where all the locks are held until the transaction commits. Thus, $S-LOCK$s and $X-LOCK$s

T_1	T_2
X-LOCK(A);	
S-LOCK(B);	
	X-LOCK(A);
read(B);	
UNLOCK(B);	
read(A);	
write(A);	
UNLOCK(A);	
COMMIT;	read(A);
	write(A);
	COMMIT;

Figura 11.22: Schedule 10. Strict 2PL without cascading aborts

are *long locks*. Strict or rigorous 2PL eliminates cascading aborts, but the schedules limit concurrency.

11.8. Deadlock handling

A deadlock occurs when there is a set of transactions where each transaction is waiting for a lock that another transaction in the set holds. No transaction in the set can progress. We can handle deadlocks by detection and recovery, or prevention.

Deadlock detection and recovery

Deadlock detection and *recovery* protocols allow the system to enter a deadlock state that will be recovered. To determine if a deadlock occured, periodically a deadlock detection algorithm is invoked and when detected the system uses recovery algorithms. If the probability of deadlock is not high, these techniques are more efficient than deadlock prevention. But, these techniques have an additional overhead of maintaining information and executing the algorithms.

Deadlocks can be detected with *waits-for graphs*. Every transaction is a node. An edge is created from T_i to T_j if T_i is waiting for transaction T_j to release a lock. The edge is removed when T_j releases the data item required by T_i. A deadlock exists in the system if, when we check periodically, there is a cycle in the wait-for graph. The deadlock detection algorithm is invoked frequently if many deadlocks occur. Worst case, the algorithm could be invoked every time a lock is not granted immediately.

An example of a wait-for graph is shown in Fig. 11.23. The subfigure 11.23a shows the schedule, while the subfigure 11.23b.

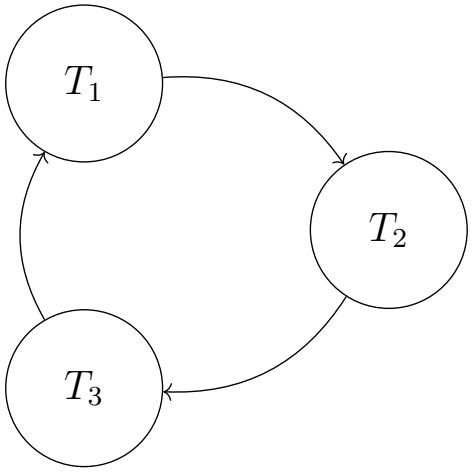
- The edge $T_1 \rightarrow T_2$ is created as T_1 waits for the $S-LOCK$ on B that T_2 holds.
- The edge $T_3 \rightarrow T_2$ is **not** created as both T_2 and T_3 can hold the $S-LOCK$ on D.
- The edge $T_2 \rightarrow T_3$ is created as T_2 waits for the $S-LOCK$ on c that T_3 holds.
- The edge $T_3 \rightarrow T_1$ is created as T_1 waits for the $X-LOCK$ on A that T_1 holds.

When a deadlock is detected, the system must *recover*. To do this, we must determine which transactions must be rolled back to break the deadlock.

The set of transactions to roll back must be selected (*victim selection*) in such a way that will incur the minimum cost. There are several factors to consider such as

T_1	T_2	T_3
S-LOCK(A);		
	X-LOCK(B);	
		X-LOCK(C);
	S-LOCK(D);	
		S-LOCK(D);
S-LOCK(B);		
	S-LOCK(C);	
		X-LOCK(A);
...	...	...

(a) Schedule with three-way deadlock



(b) Wait-for graph to the schedule

Figura 11.23: Deadlock represented in a wait-for graph

the age, progress, number of data items used and number of transactions involved in the rollback. A combination of several factors are considered. We must include the number of rollbacks as a cost factor to not select only one victim.

When we decide the victim transaction, it has to determine how far the transaction needs to be rolled back. A *total rollback* aborts all the transaction and restarts it. A *partial rollback*, using the sequence of grants and requests of locks with the updates, can determine a point to revert the changes to resume the execution from there.

Deadlock prevention

Deadlock prevention protocols ensure that the system will never enter a deadlock state. These schemes are used if the probability of deadlock is high. Several approaches have been proposed.

Avoiding cyclical waits approaches

The first approach focuses on not allowing cyclical waits. The simplest way to achieve this is to acquire all the locks at the beginning of the transaction execution in one step, but the data-item utilization is low and is difficult to predict before initialization. Furthermore, when the transaction cannot acquire all the locks it must wait. An example can be seen in Fig. 11.24 where T_2 must wait as it could not acquire all the locks, while T_1 has all the locks. T_2 can only continue when all the locks required are released.

T_1	T_2
X-LOCK(A,B);	
...	**X-LOCK(A,B);**
UNLOCK(A);	
...	
UNLOCK(B);	
COMMIT;	...

Figura 11.24: Deadlock prevention acquiring all locks in one step

Another one of these approaches consists of ordering all the data items and strictly following the ordering. A variation considering both of these schemes is to use 2PL

with strict ordering, thus the locks can only be requested following the order. An example can be seen in Fig. 11.25. The locks are defined to be acquired with A first and then B. A deadlock does not occur when T_2 tries to acquire A, the transaction must wait. If we had no order, T_2 could acquire the $X - LOCK(B)$ first and a deadlock would happen.

T_1	T_2
X-LOCK(A);	**X-LOCK(A);**
X-LOCK(B);	
...	
UNLOCK(B);	
UNLOCK(A);	
COMMIT;	**X-LOCK(B);**
	...

Figura 11.25: Deadlock prevention 2PL with strict ordering

Roll back approaches

Furthermore, there is a roll back approach. When a transaction tries to acquire a lock held by another transaction (possible deadlock), one of the transactions will be rolled back. To decide which transaction to rollback a unique timestamp is assigned to each transaction when it begins, prioritizing older transactions. The rolled back transactions retain it's initial timestamp. There are two different timestamp-based schemes:

- **Wait-die ("Old waits for young"):** If the requesting transaction has a higher priority than the holding transaction, it waits. Otherwise, it is aborted. The older transaction is allowed to *wait* for a younger transaction. The younger transaction *dies* (aborts) if it requests the lock held by an older transaction.
- **Wound-wait ("Young waits for old"):** If the requesting transaction has a higher priority than the holding transaction, the holding transaction aborts and rolls back (wounds). Otherwise, it waits. The younger transaction is allowed to *wait* for an older transaction. The older transaction *wounds* (abort) the younger transaction holding the lock.

An example for both timestamped-based schemes is shown in Fig. 11.26. For example in Subfigure. 11.26a T_1 starts before T_2, thus $T_1 < T_2$ (T_1 is older). When T_1 requests a lock held by T_2, for wait-die the older transaction T_1 waits for transaction T_2 to release the lock. If the protocol was wound-wait then T_2 is aborted and rolled back. For another example shown in Subfigure. 11.26a T_1 also starts before T_2 (T_1 is older). With the wait-die scheme when T_2 requests the lock held by T_1, T_2 dies. While, for the wound-die scheme T_2 can wait for the older transaction.

Both timestamped-based techniques prevent deadlocks as either transactions wait for the younger (wait-die) or older transactions (wound-wait), therefore no cycle is created. However, both techniques have many unnecessary rollbacks.

Another method is using *lock timeouts*, defining a specified amount for waiting. If the transaction was not granted the lock after waiting the specified time, it will roll back and restart. Defining the time to wait is difficult, long times cause unnecessary delays while short waits lead to wasted results. This protocol falls between deadlock prevention and detection. Starvation can occur, thus there is limited applicability to the scheme.

T_1	T_2
BEGIN	
	BEGIN
	X-LOCK(A);
X-LOCK(A);	

(a) Wait-die T_1 waits and wound-wait T_2 aborts

T_1	T_2
BEGIN	
X-LOCK(A);	
	BEGIN
	X-LOCK(A);

(b) Wait-die T_2 waits and wound-wait T_1 aborts

Figura 11.26: Deadlock prevention based on timestamps

11.9. Multiple granularity

Locking many single data items is costly, as it requires the lock manager to request many locks. However, instead of only locking a single data item, we could consider different types of *granularity*. This way *smaller* or *finer* granularities, such as attributes, can be requested by locking a *larger* or *coarser* granularity item, such as a table. Large granularities allow for less concurrency, while smaller granularities have a higher overhead with the lock manager. The level of granularity used will depend on the type of transactions involved and will be selected by the DBMS.

The size and hierarchy of the different data granularity can be defined as a tree. An example of such a tree is shown in Fig. 11.27. Every node is an independent data item. The highest level is all the databases. The second level is the tables saved in the database. The third level are tuples of the respective table. Lastly, the fourth level represents the attributes of the respective tuple. If we acquire a lock on Table 1 (*explicit lock*), then all the descendants such as Tuple 1, Tuple 2, $\cdots$, Tuple, Attribute 1, Attribute 2, $\cdots$ and Attribute p will acquire the same lock (*implicit lock*). The granularity hierarchy tree could also have the following four levels: database, areas, files and records.

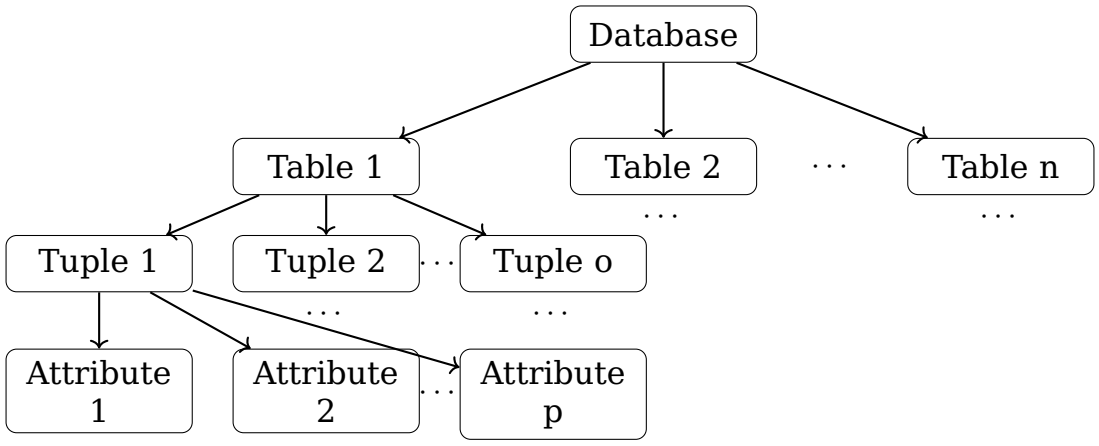


Figura 11.27: Granularity hierarchy

To determine if a lock can be granted we have to check if all the implicit locks are compatible with requested mode. To determine this, all the descendants of the granularity level must be checked. However, this is not efficient. In the worst case, locking the database requires checking all the nodes in the tree.

To check more efficiently, we can use *intention lock modes* to not have to check all descendant nodes. With intention mode, all the descendants are explicitly locked. The different modes allowed are the following:

- **Intention-shared (IS) mode:** Indicares that the descendants have explicit shared-mode locks.

- **Intention-exclusive (IX) mode:** Indicates that the descendants have explicit exclusive-mode or shared-mode locks.

- **Shared and intention-exclusive (SIX) mode:** The node is explicitly locked in shared-mode, with the descendants explicitly locked in exclusive-mode.

The compilability matrix of the modes are shown in Fig. 11.28.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

Figura 11.28: Compatibility matrix of intention lock modes

The *multiple-granularity locking protocol* can ensure serializability by setting the lock at the highest level of the database hierarchy. For example, assume a hierarchy tree for a database that saves Students' records. We will only exemplify the protocol using the second (table) and third level (tuple) of the hierarchy tree. The final results of the locks are shown in Fig. 11.29.

- Suppose that transaction T_1 wants to read the data for Alice. Therefore, an $S - LOCK$ is acquired for the tuple with Alice's data and a $IS - LOCK$ on the Student table indicating that one of its descendants (Alice's tuple) has an $S - LOCK$.

- Now, let us assume that transaction T_2 wants to update Carlos's record. We would require to acquire an $X - LOCK$ for Carlos' tuple and an $IX - LOCK$ on the Student table. We can acquire the lock on the table, as the $IS - LOCK$ is compatible with the $IX - LOCK$.

- Finally, let's assume that transaction T_3 scans wants to scan the student table to update some student records. This would require gaining a $SIX - LOCK$ on the student table. However, this would not be possible as SIX is not compatible with IX. Therefore, T_3 would have to wait to be granted the lock. If T_3 happend before T_2, we could have acquired the lock for T_3.

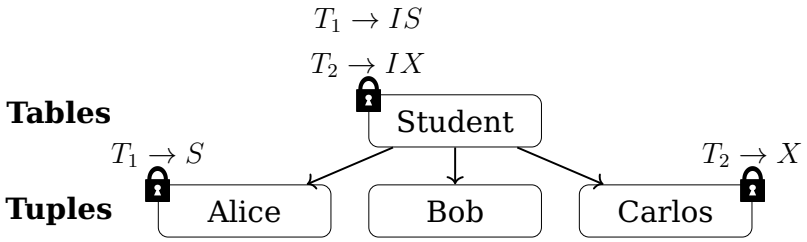


Figura 11.29: Result of applying the multiple granularity protocol

Locks must be acquired from the root to the leaf, and released from the left to the root. This protocol enhances concurrency and reduces overhead, but deadlocks can still occur.

11.10. Timestamp-based protocols

The lock based protocols define the order at execution time, but we could also define the serializability order in advance using a *timestamp-ordering scheme*. A unique timestamp $TS(T_i)$ for every transaction T_i before execution.

When a new transaction T_j enters the system then $TS(T_i) < TS(T_j)$. To assign the sequential timestamp we could use different strategies. The *system clock* can be used to assign a value when the transaction enters the system, however there can be issues in edge cases (e.g., daylight savings). Another option is to use a *logical counter* that increments after a new transaction enters the system, but the counter could overflow or has issues maintaining the counter across multiple machines. A hybrid combination of methods can be used. If $TS(T_i) < TS(T_j)$ the DBMS must ensure an equivalent serial execution of T_i appearing before T_j.

The *timestamp-ordering protocol* (basic T/O) executes conflicting $reads$ and $write$ operations in the timestamp order, ensuring conflict serializability. To execute the scheme, the DBMS tracks for every data item Q the last transaction that executed successfully a $read$ ($R - TS(Q)$) and a $write$ ($W - TS(Q)$). The DBMS updates these timestamps after every instruction is executed.

- **Read operations.**
  - If $TS(T_i) < W - TS(Q)$, a future transaction has written to data item Q before T_i violating the $TS(T_i) < TS(T_j)$ property. Therefore, T_i is aborted and restarted with a new timestamp value.
  - Else, $TS(T_i) \geq W - TS(Q)$ the order $TS(T_i) < TS(T_j)$ is preserved and the $read(Q)$ instruction is executed. The DBMS also updates $R - TS(Q) = max(R - TS(Q), TS(T_i))$.
- **Write operations.**
  - If $TS(T_i) < R - TS(Q)$ or $TS(T_i) < W - TS(Q)$, a future transaction has read or written to data item Q before T_i violating the $TS(T_i) < TS(T_j)$ property. Therefore, T_i is aborted and restarted with a new timestamp value.
  - Else, $TS(T_i) \geq R - TS(Q)$ and $TS(T_i) \geq W - TS(Q)$ the order of execution is ensured and the $write(Q)$ instruction is executed. The DBMS also updates $W - TS(Q) = TS(T_i)$.

The timestamped protocol example will use the schedule in Fig. 11.30. We can assume that $TS(T_1) = 1$ and $TS(T_2) = 2$.

T_1	T_2
read(B);	
	read(B);
	write(B);
read(A);	
	read(A);
read(A);	
	write(A);

Figura 11.30: Schedule for basic T/O example

- When T_1 executes $read(B)$, $W - TS(B) = 0$. As$TS(T_1) \geq W - TS(B) = 1 \geq 0$, T_1 can execute the instruction and $R - TS(B) = max(R - TS(B), TS(T_1)) = max(0, 1) = 1$.

- When T_2 executes $read(B)$, $W - TS(B) = 0$. As $TS(T_2) \geq W - TS(B) = 2 \geq 0$, T_2 can execute the instruction and $R - TS(B) = max(R - TS(B), TS(T_2)) = max(1, 2) = 2$.
- When T_2 executes $write(B)$, $W - TS(B) = 0$ and $R - TS(B) = 2$. As $TS(T_2) \geq W - TS(B) = 2 \geq 0$ and $TS(T_2) \geq R - TS(B) = 2 \geq 2$, T_2 can execute the instruction and $W - TS(B) = TS(T_2) = 2$.
- When T_1 executes $read(A)$, $W - TS(A) = 0$. As $TS(T_1) \geq W - TS(A) = 1 \geq 0$, T_1 can execute the instruction and $R - TS(A) = max(R - TS(A), TS(T_1)) = max(0, 1) = 1$.
- When T_2 executes $read(A)$, $W - TS(A) = 0$. As $TS(T_2) \geq W - TS(A) = 2 \geq 0$, T_2 can execute the instruction and $R - TS(A) = max(R - TS(A), TS(T_2)) = max(1, 2) = 2$.
- When T_1 executes $read(A)$, $W - TS(A) = 0$. As $TS(T_1) \geq W - TS(A) = 1 \geq 0$, T_1 can execute the instruction and $R - TS(A) = max(R - TS(A), TS(T_1)) = max(2, 1) = 2$.
- When T_2 executes $write(A)$, $W - TS(A) = 0$ and $R - TS(A) = 2$. As $TS(T_2) \geq W - TS(A) = 2 \geq 0$ and $TS(T_2) \geq R - TS(a) = 2 \geq 2$, T_2 can execute the instruction and $W - TS(A) = TS(T_2) = 2$.

The protocol can also be modified with *Thomas' write rule* to remove unnecessary rollbacks for write, creating view serializable conflicts. If $TS(T_i) < W - TS(Q)$ the system can ignore the write as it is obsolete. This does violate the timestamp order, but it is fine as no other transaction will read the $write(Q)$ of T_i.

The protocol ensures conflict serializability and freedom from deadlocks, though it has issues due to starvation, non-recoverable schedules and phantom tuples. Some of these problems may be fixed with variations of the protocol.

11.11. Validation-based protocols

When most transactions only invoke $read$ the conflicts are low, thus the concurrency control schemes may provide unnecessary overhead. When a transaction reads a value integrity is never lost, thus only writing will we require *validating* the consistency of the database. There are three phases of execution for writes, shown in Fig. 11.31, that must be executed sequentially for a transaction to ensure the consistency.

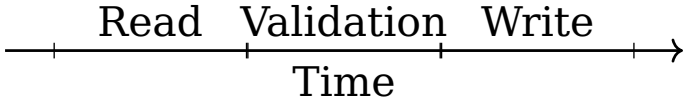


Figura 11.31: Validation protocol transaction execution phases (only writes)

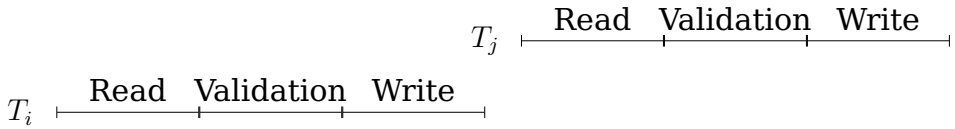
1. **Read phase.** The data is *read* to a temporary copy. The transaction the modified the temporary results (not global).
2. **Validation phase.** The system *validates* whether a serializability conflict occurred. If there is a violation the transaction is aborted.
3. **Write phase.** The temporary results are *writen* to the database (global).

To validate the serial equivalence, each transaction T_i will be assigned an order of execution based on a timestamp $TS(T_i)$. If T_i is older than T_j then $TS(T_i) < TS(T_j)$. To ensure the equivalence the serial equivalence for any younger transaction T_j for every older transaction T_i one of following conditions must hold (Fig. 11.32):

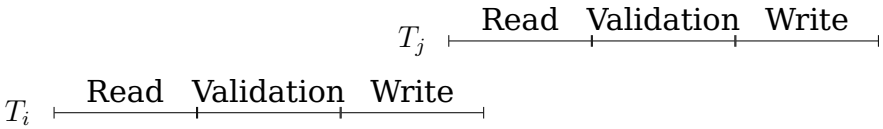
1. T_i completes all phases before T_j begins (Subfig. 11.32a).

2. T_i completes all phases before T_j enters the validation phase and the transactions do not read the same items (Subfig. 11.32b).

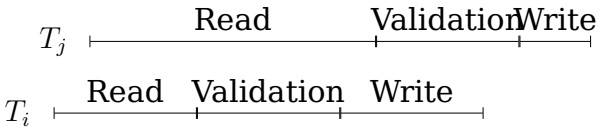
3. T_i completes the read phase before T_j and the transactions do not read or write the same items (Subfig. 11.32c).



(a) T_i finishes before T_j begins



(b) T_i finishes before T_j enters validation



(c) T_i completes the read phase before T_j

Figura 11.32: Validation order of transactions for validation protocol

This approach is optimistic as the validation will fail only in the worst of cases. The protocol automatically guards against rollbacks and has no deadlocks, however starvation can happen.

11.12. Isolation levels

Programmers can decide the appropriate level of concurrency required. *Isolation levels* control the extent of isolation between transactions. Depending on the application, a weaker isolation level may improve the system performance but increases the risk of inconsistency.

Different DBMS implement different isolation levels, however most of them are based from the *SQL-92 standard* that defines the following isolation levels based on transaction problems: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The degree of transaction concurrency versus database consistency for each isolation level is shown in Fig. 11.33.

The transaction problems allowed by isolation level are shown in Fig. 11.34. An x-mark (✗) indicates that the problem cannot occur, while a question mark (?) indicates that it might happen.

The SQL-92 isolation levels can be implemented using 2PL-lock based systems. The length of holding each lock is shown in Fig. 11.35. All isolation levels hold long $X-LOCK$s to ensure that no *lost update* issues occur. While depending on the isolation level the length and type of lock held for $S-LOCK$s vary. Locks for data-items holds the particular data-item, while locks for a condition lock all elements that satisfy the condition. Not all DBMS isolation levels that are not analogous to locking protocols, thus new systems may consider other concurrency management protocols.

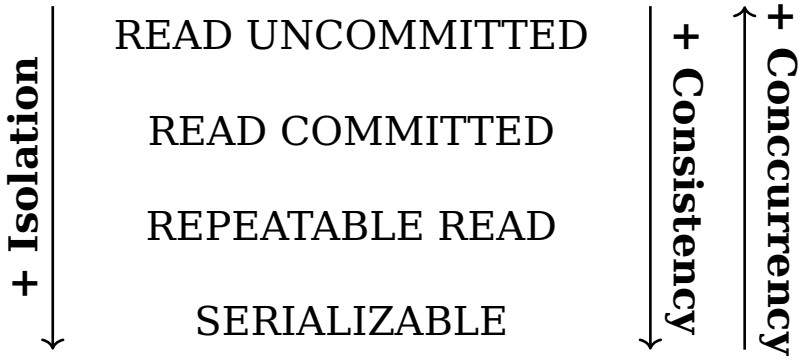


Figura 11.33: Transaction concurrency versus database consistency with SQL-92 isolation levels

	Lost update	Dirty read	Unrepeatable read	Phantom read
READ UNCOMMITTED	✗	?	?	?
READ COMMITTED	✗	✗	?	?
REPEATABLE READ	✗	✗	✗	?
SERIALIZABLE	✗	✗	✗	✗

Figura 11.34: Isolation levels with transaction problems allowed

	$S-LOCK$		$X-LOCK$
	data-item	condition	
READ UNCOMMITTED	None	None	Long
READ COMMITTED	Short	Short	Long
REPEATABLE READ	Long	Short	Long
SERIALIZABLE	Long	Long	Long

Figura 11.35: Lock liberation for isolation levels

Most DBMS run the **READ COMMITTED** isolation level by default. We can change the isolation level in *SQL SERVER* with the **SET TRANSACTION ISOLATION LEVEL** <**NAME**>. Furthermore, by default most DBMS commit every statement after execution (*automatic commit*). To enable manual commits, the transaction begins with the start transaction command (in SQL Server it starts with **BEGIN TRANSACTION**) and ends with a **COMMIT** or **ROLLBACK**.

11.13. Review

Problemas de control de concurrencia

1. Se tiene el siguiente *schedule*:

T_1	T_2
read(A);	
write(C);	
	write(C);

   (a) ¿Cuáles de los siguientes problemas de control de concurrencia tiene el *schedule*? Marque todas las opciones posibles. ◯ Lost update ◯ Dirty read ◯ Unrepeatable read ◯ Phantom read

   (b) Para cada uno de los problemas anteriores, detalle todas las instrucciones que generan cada problema respectivo. Debe indicar para cual transacción genera el problema respectivo.

   (c) Si se quisieran evitar todos los problemas de control de concurrencia de este, ¿cuál es el nivel de aislamiento (*isolation level*) mínimo que se debe seleccionar? ◯ Read uncommitted ◯ Read committed ◯ Repeatable read. ◯ Serializable.

   (d) Explique su escogencia del nivel de aislamiento.

2. Se tiene el siguiente *schedule*:

T_1	T_2
read(A);	
	write(B);
write(C);	
	read(D);

   (a) ¿Cuáles de los siguientes problemas de control de concurrencia tiene el *schedule*? Marque todas las opciones posibles. ◯ Lost update ◯ Dirty read ◯ Unrepeatable read ◯ Phantom read

   (b) Para cada uno de los problemas anteriores, detalle todas las instrucciones que generan cada problema respectivo. Debe indicar para cual transacción genera el problema respectivo.

   (c) Si se quisieran evitar todos los problemas de control de concurrencia de este, ¿cuál es el nivel de aislamiento (*isolation level*) mínimo que se debe seleccionar? ◯ Read uncommitted ◯ Read committed ◯ Repeatable read. ◯ Serializable.

   (d) Explique su escogencia del nivel de aislamiento.

T_1	T_2
read(A);	
	read(A);
read(A);	
	read(B);
read(A);	
	read(A);
write(B);	

3. Se tiene el siguiente *schedule*:

   (a) ¿Cuáles de los siguientes problemas de control de concurrencia tiene el *schedule*? Marque todas las opciones posibles. ◯ Lost update ◯ Dirty read ◯ Unrepeatable read ◯ Phantom read

   (b) Para cada uno de los problemas anteriores, detalle todas las instrucciones que generan cada problema respectivo. Debe indicar para cual transacción genera el problema respectivo.

   (c) Si se quisieran evitar todos los problemas de control de concurrencia de este, ¿cuál es el nivel de aislamiento (*isolation level*) mínimo que se debe seleccionar? ◯ Read uncommitted ◯ Read committed ◯ Repeatable read. ◯ Serializable.

   (d) Explique su escogencia del nivel de aislamiento.

4. Se tiene el siguiente *schedule*:

T_1	T_2	T_3	T_4
read(A);			
	write(B);		
			read(D);
		write(D);	
		write(C);	
			read(C);
	write(C);		

   (a) ¿Cuáles de los siguientes problemas de control de concurrencia tiene el *schedule*? Marque todas las opciones posibles. ◯ Lost update ◯ Dirty read ◯ Unrepeatable read ◯ Phantom read

   (b) Para cada uno de los problemas anteriores, detalle todas las instrucciones que generan cada problema respectivo. Debe indicar para cual transacción genera el problema respectivo.

   (c) Si se quisieran evitar todos los problemas de control de concurrencia de este, ¿cuál es el nivel de aislamiento (*isolation level*) mínimo que se debe seleccionar? ◯ Read uncommitted ◯ Read committed ◯ Repeatable read. ◯ Serializable.

   (d) Explique su escogencia del nivel de aislamiento.

5. Se tiene el siguiente *schedule*:

   (a) ¿Cuáles de los siguientes problemas de control de concurrencia tiene el *schedule*? Marque todas las opciones posibles. ◯ Lost update ◯ Dirty read ◯ Unrepeatable read ◯ Phantom read

T_1	T_2	T_3
read(B);		
		read(C);
write(B);		
		read(A);
write(A);		
		write(C);
	read(C);	
	write(C);	
		write(A);
read(A);		
	write(B);	

(b) Para cada uno de los problemas anteriores, detalle todas las instrucciones que generan cada problema respectivo. Debe indicar para cual transacción genera el problema respectivo.

(c) Si se quisieran evitar todos los problemas de control de concurrencia de este, ¿cuál es el nivel de aislamiento (*isolation level*) mínimo que se debe seleccionar? ◯ Read uncommitted ◯ Read committed ◯ Repeatable read. ◯ Serializable.

(d) Explique su escogencia del nivel de aislamiento.

6. Se tiene el siguiente *schedule*:

T_1	T_2	T_3
	write(A);	
		read(B);
write(C);		
	read(A);	
read(B);		
		write(B);
	read(C);	
	write(C);	
		write(A);
read(B);		

(a) ¿Cuáles de los siguientes problemas de control de concurrencia tiene el *schedule*? Marque todas las opciones posibles. ◯ Lost update ◯ Dirty read ◯ Unrepeatable read ◯ Phantom read

(b) Para cada uno de los problemas anteriores, detalle todas las instrucciones que generan cada problema respectivo. Debe indicar para cual transacción genera el problema respectivo.

(c) Si se quisieran evitar todos los problemas de control de concurrencia de este, ¿cuál es el nivel de aislamiento (*isolation level*) mínimo que se debe seleccionar? ◯ Read uncommitted ◯ Read committed ◯ Repeatable read. ◯ Serializable.

(d) Explique su escogencia del nivel de aislamiento.

Protocolos de control de concurrencia

Para cada uno de los siguientes protocolos, seleccione **todos** los errores que pueden suceder.

1. Utilizando *candados básicos* puede suceder:

   ◯ Inconsistencia en estados de BD ◯ *Starvation* ◯ *Deadlocks* ◯ Concurrencia limitada ◯ *Cascading aborts*

2. Utilizando *2PL* puede suceder:

   ◯ Inconsistencia en estados de BD ◯ *Starvation* ◯ *Deadlocks* ◯ Concurrencia limitada ◯ *Cascading aborts*

3. Utilizando *strict or rigorous 2PL* puede suceder:

   ◯ Inconsistencia en estados de BD ◯ *Starvation* ◯ *Deadlocks* ◯ Concurrencia limitada ◯ *Cascading aborts*

4. Utilizando *granularidad múltiple* puede suceder:

   ◯ Inconsistencia en estados de BD ◯ *Starvation* ◯ *Deadlocks* ◯ Concurrencia limitada ◯ *Cascading aborts*

5. Utilizando *basic T/O* puede suceder:

   ◯ Inconsistencia en estados de BD ◯ *Starvation* ◯ *Deadlocks* ◯ Concurrencia limitada ◯ *Cascading aborts*

6. Utilizando *validation* puede suceder:

   ◯ Inconsistencia en estados de BD ◯ *Starvation* ◯ *Deadlocks* ◯ Concurrencia limitada ◯ *Cascading aborts*

Serializabilidad

Para cada uno de los siguientes *schedules*, responda las preguntas.

1. Se tiene el siguiente *schedule*:

T_1	T_2
read(A);	
write(C);	
	write(C);

   (a) ¿El *schedule* es serial? ◯ Si ◯ No

   (b) Dibuje el gráfico de dependencias del *schedule*.

   (c) ¿El *schedule* es serializable en conflictos (*conflict serializable*)? ◯ Si ◯ No

   (d) Si su respuesta a la pregunta anterior es "Si", provea el ejecución serial equivalente. En caso contrario provea los ciclos que indican que no es serializable en conflictos.

   (e) Seleccione todas las transacciones que se deben remover para hacer que la transacción sea serializable en conflictos.

   ◯ T_1 ◯ T_2 ◯ Ya es serializable en conflictos.

T_1	T_2
read(A);	
	write(B);
write(C);	
	read(D);

2. Se tiene el siguiente *schedule*:
   (a) ¿El *schedule* es serial? ◯ Si ◯ No
   (b) Dibuje el gráfico de dependencias del *schedule*.
   (c) ¿El *schedule* es serializable en conflictos (*conflict serializable*)? ◯ Si ◯ No
   (d) Si su respuesta a la pregunta anterior es "Si", provea el ejecución serial equivalente. En caso contrario provea los ciclos que indican que no es serializable en conflictos.
   (e) Seleccione todas las transacciones que se deben remover para hacer que la transacción sea serializable en conflictos.
   ◯ T_1 ◯ T_2 ◯ Ya es serializable en conflictos.
3. Se tiene el siguiente *schedule*:

T_1	T_2
read(A);	
	read(A);
read(A);	
	read(B);
read(A);	
	read(A);
write(B);	

   (a) ¿El *schedule* es serial? ◯ Si ◯ No
   (b) Dibuje el gráfico de dependencias del *schedule*.
   (c) ¿El *schedule* es serializable en conflictos (*conflict serializable*)? ◯ Si ◯ No
   (d) Si su respuesta a la pregunta anterior es "Si", provea el ejecución serial equivalente. En caso contrario provea los ciclos que indican que no es serializable en conflictos.
   (e) Seleccione todas las transacciones que se deben remover para hacer que la transacción sea serializable en conflictos.
   ◯ T_1 ◯ T_2 ◯ Ya es serializable en conflictos.
4. Se tiene el siguiente *schedule*:
   (a) ¿El *schedule* es serial? ◯ Si ◯ No
   (b) Dibuje el gráfico de dependencias del *schedule*.
   (c) ¿El *schedule* es serializable en conflictos (*conflict serializable*)? ◯ Si ◯ No
   (d) Si su respuesta a la pregunta anterior es "Si", provea el ejecución serial equivalente. En caso contrario provea los ciclos que indican que no es serializable en conflictos.

T_1	T_2	T_3	T_4
read(A);			
	write(B);		
			read(D);
		write(D);	
		write(C);	
			read(C);
	write(C);		

(e) Seleccione todas las transacciones que se deben remover para hacer que la transacción sea serializable en conflictos.
◯ T_1 ◯ T_2 ◯ T_3 ◯ T_4 ◯ Ya es serializable en conflictos.

5. Se tiene el siguiente *schedule*:

T_1	T_2	T_3
read(B);		
		read(C);
write(B);		
		read(A);
write(A);		
		write(C);
	read(C);	
	write(C);	
		write(A);
read(A);		
	write(B);	

(a) ¿El *schedule* es serial? ◯ Si ◯ No

(b) Dibuje el gráfico de dependencias del *schedule*.

(c) ¿El *schedule* es serializable en conflictos (*conflict serializable*)? ◯ Si ◯ No

(d) Si su respuesta a la pregunta anterior es "Si", provea el ejecución serial equivalente. En caso contrario provea los ciclos que indican que no es serializable en conflictos.

(e) Seleccione todas las transacciones que se deben remover para hacer que la transacción sea serializable en conflictos.
◯ T_1 ◯ T_2 ◯ T_3 ◯ Ya es serializable en conflictos.

6. Se tiene el siguiente *schedule*:

(a) ¿El *schedule* es serial? ◯ Si ◯ No

(b) Dibuje el gráfico de dependencias del *schedule*.

(c) ¿El *schedule* es serializable en conflictos (*conflict serializable*)? ◯ Si ◯ No

(d) Explique formalmente porque es o no es un *schedule* serializable en conflictos.

(e) Seleccione todas las transacciones que se deben remover para hacer que la transacción sea serializable en conflictos.
◯ T_1 ◯ T_2 ◯ T_3 ◯ Ya es serializable en conflictos.

T_1	T_2	T_3
	write(A);	
		read(B);
write(C);		
	read(A);	
read(B);		
		write(B);
	read(C);	
	write(C);	
		write(A);
read(B);		

Candados

Para cada una de los siguientes *schedules*, indique que candado simple se debe pedir. Asuma que despues de obtener un candado, no se libera.

1. Se tiene el siguiente schedule:

	T_1	T_2	T_3
t_1	$read(A)$;		
t_2		$read(B)$;	
t_3			$write(A)$;
t_4		$write(B)$;	
t_5			$write(C)$;
t_6	$read(A)$;		

   (a) En t_1 se pide: ◯ $S-LOCK(A)$ ◯ $X-LOCK(A)$ ◯ $S-LOCK(B)$ ◯ $X-LOCK(B)$ ◯ $S-LOCK(C)$ ◯ $X-LOCK(C)$ ◯ Nada

   (b) En t_2 se pide: ◯ $S-LOCK(A)$ ◯ $X-LOCK(A)$ ◯ $S-LOCK(B)$ ◯ $X-LOCK(B)$ ◯ $S-LOCK(C)$ ◯ $X-LOCK(C)$ ◯ Nada

   (c) En t_3 se pide: ◯ $S-LOCK(A)$ ◯ $X-LOCK(A)$ ◯ $S-LOCK(B)$ ◯ $X-LOCK(B)$ ◯ $S-LOCK(C)$ ◯ $X-LOCK(C)$ ◯ Nada

   (d) En t_4 se pide: ◯ $S-LOCK(A)$ ◯ $X-LOCK(A)$ ◯ $S-LOCK(B)$ ◯ $X-LOCK(B)$ ◯ $S-LOCK(C)$ ◯ $X-LOCK(C)$ ◯ Nada

   (e) En t_5 se pide: ◯ $S-LOCK(A)$ ◯ $X-LOCK(A)$ ◯ $S-LOCK(B)$ ◯ $X-LOCK(B)$ ◯ $S-LOCK(C)$ ◯ $X-LOCK(C)$ ◯ Nada

   (f) En t_6 se pide: ◯ $S-LOCK(A)$ ◯ $X-LOCK(A)$ ◯ $S-LOCK(B)$ ◯ $X-LOCK(B)$ ◯ $S-LOCK(C)$ ◯ $X-LOCK(C)$ ◯ Nada

2. Se tiene el siguiente schedule:

   Indique para cada instrucción que candado simple se debe pedir. Debe indicar una de las siguientes opciones en cada espacio: $S-LOCK(A)$, $X-LOCK(A)$, $S-LOCK(B)$, $X-LOCK(B)$, $S-LOCK(C)$, $X-LOCK(C)$ y Nada. Asuma que después de obtener un candado, no se libera. La opción de nada indica que no hay que pedir un candado.

Manejo de deadlocks: Detección

1. Se tiene las siguientes transacciones que piden candados simples:

	T_1	T_2	T_3
t_1		write(A);	
t_2			read(B);
t_3	write(C);		
t_4		read(A);	
t_5	read(B);		
t_6			write(B);
t_7		read(C);	
t_8		write(C);	
t_9			write(A);
t_{10}	read(B);		

	T_1	T_2	T_3
t_1		_____	
t_2			_____
t_3	_____		
t_4		_____	
t_5	_____		
t_6			_____
t_7		_____	
t_8		_____	
t_9			_____
t_{10}	_____		

(a) Para cada tiempo, indique si cada candado otorgará (*granted*) o bloqueará (*blocked*) el candado.

i. En t_1: ◯ Otorgará ◯ Bloqueará
ii. En t_2: ◯ Otorgará ◯ Bloqueará
iii. En t_3: ◯ Otorgará ◯ Bloqueará
iv. En t_4: ◯ Otorgará ◯ Bloqueará
v. En t_5: ◯ Otorgará ◯ Bloqueará
vi. En t_6: ◯ Otorgará ◯ Bloqueará
vii. En t_7: ◯ Otorgará ◯ Bloqueará
viii. En t_8: ◯ Otorgará ◯ Bloqueará

(b) Para las transacciones anteriores, dibuje el *wait-for graph*.

(c) ¿Existe un deadlock en el *schedule* anterior? ◯ Si ◯ No

(d) Explique porque hay o no hay un deadlock.

2. Se tiene las siguientes transacciones que piden candados simples:

(a) Para el *schedule* anterior, dibuje el *wait-for graph*.

(b) ¿Existe un deadlock en el *schedule* anterior? ◯ Si ◯ No

(c) Explique porque hay o no hay un deadlock.

Manejo de deadlocks: Prevención

Para cada uno de los siguientes protocolos, indique si se otorga el candado (O), bloquea el candado o si se encuentra la transacción bloqueada(B), aborta una transacción (A) o se encuentra muerta la transacción (M). Asuma que después de obtener un candado, no se libera. Además asuma que las transacciones se crearon de tal manera que las estampillas de tiempo son: $T_1 < T_2 < T_3$.

	T_1	T_2	T_3
t_1	**S-LOCK(A);**		
t_2		**X-LOCK(B);**	
t_3			**X-LOCK(C);**
t_4		**S-LOCK(A);**	
t_5		**X-LOCK(C);**	
t_6			**X-LOCK(A);**
t_7	**S-LOCK(D);**		
t_8	**S-LOCK(A);**		

	T_1	T_2	T_3
t_1		$X-LOCK(A)$	
t_2			$S-LOCK(B)$
t_3	$X-LOCK(C)$		
t_4		Nada	
t_5	$S-LOCK(B)$		
t_6			$X-LOCK(B)$
t_7		$S-LOCK(C)$	
t_8		$X-LOCK(C)$	
t_9			$X-LOCK(A)$
t_{10}	Nada		

1. Se tiene las siguientes llamadas a candados:

	T_1	T_2	T_3
t_1	**S-LOCK(C);**		
t_2			**X-LOCK(C);**
t_3		**X-LOCK(A);**	
t_4		**S-LOCK(B);**	
t_5	**X-LOCK(A);**		
t_6			**X-LOCK(A);**
t_7			**S-LOCK(B);**

   (a) Si *no* se utiliza alguna política de prevención:
      i. En t_1: ◯ O ◯ B ◯ A ◯ M
      ii. En t_2: ◯ O ◯ B ◯ A ◯ M
      iii. En t_3: ◯ O ◯ B ◯ A ◯ M
      iv. En t_4: ◯ O ◯ B ◯ A ◯ M
      v. En t_5: ◯ O ◯ B ◯ A ◯ M
      vi. En t_6: ◯ O ◯ B ◯ A ◯ M
      vii. En t_7: ◯ O ◯ B ◯ A ◯ M

   (b) Si se utiliza la política *wait-die*:
      i. En t_1: ◯ O ◯ B ◯ A ◯ M
      ii. En t_2: ◯ O ◯ B ◯ A ◯ M
      iii. En t_3: ◯ O ◯ B ◯ A ◯ M
      iv. En t_4: ◯ O ◯ B ◯ A ◯ M
      v. En t_5: ◯ O ◯ B ◯ A ◯ M
      vi. En t_6: ◯ O ◯ B ◯ A ◯ M
      vii. En t_7: ◯ O ◯ B ◯ A ◯ M

(c) Si se utiliza la política *wound-wait*:

i. En t_1: ◯ O ◯ B ◯ A ◯ M
ii. En t_2: ◯ O ◯ B ◯ A ◯ M
iii. En t_3: ◯ O ◯ B ◯ A ◯ M
iv. En t_4: ◯ O ◯ B ◯ A ◯ M
v. En t_5: ◯ O ◯ B ◯ A ◯ M
vi. En t_6: ◯ O ◯ B ◯ A ◯ M
vii. En t_7: ◯ O ◯ B ◯ A ◯ M

2. Se tienen las siguientes llamadas a candados:

	T_1	T_2	T_3
t_1		$X - LOCK(A)$	
t_2			$S - LOCK(B)$
t_3	$X - LOCK(C)$		
t_4		Nada	
t_5	$S - LOCK(B)$		
t_6			$X - LOCK(B)$
t_7		$S - LOCK(C)$	
t_8		$X - LOCK(C)$	
t_9			$X - LOCK(A)$
t_{10}	Nada		

(a) Si *no* se utiliza alguna política de prevención:

i. En t_5: ◯ O ◯ B ◯ A ◯ M ◯ N
ii. En t_6: ◯ O ◯ B ◯ A ◯ M ◯ N
iii. En t_7: ◯ O ◯ B ◯ A ◯ M ◯ N
iv. En t_8: ◯ O ◯ B ◯ A ◯ M ◯ N
v. En t_9: ◯ O ◯ B ◯ A ◯ M ◯ N
vi. En t_{10}: ◯ O ◯ B ◯ A ◯ M ◯ N

(b) Si se utiliza la política *wait-die*:

i. En t_5: ◯ O ◯ B ◯ A ◯ M ◯ N
ii. En t_6: ◯ O ◯ B ◯ A ◯ M ◯ N
iii. En t_7: ◯ O ◯ B ◯ A ◯ M ◯ N
iv. En t_8: ◯ O ◯ B ◯ A ◯ M ◯ N
v. En t_9: ◯ O ◯ B ◯ A ◯ M ◯ N
vi. En t_{10}: ◯ O ◯ B ◯ A ◯ M ◯ N

(c) Si se utiliza la política *wound-wait*:

i. En t_5: ◯ O ◯ B ◯ A ◯ M ◯ N
ii. En t_6: ◯ O ◯ B ◯ A ◯ M ◯ N
iii. En t_7: ◯ O ◯ B ◯ A ◯ M ◯ N
iv. En t_8: ◯ O ◯ B ◯ A ◯ M ◯ N
v. En t_9: ◯ O ◯ B ◯ A ◯ M ◯ N
vi. En t_{10}: ◯ O ◯ B ◯ A ◯ M ◯ N

Bibliografía

[1] R. Elmasri and S. Navathe, *Fundamentals of database systems*, 7th ed. Pearson, 2016.

[2] A. Crotty and L. Ma. Database systems. [Online]. Available: https://15445.courses.cs.cmu.edu/fall2021/schedule.html

[3] C. Faloutsos and A. Pavlo. [Online]. Available: https://15415.courses.cs.cmu.edu/fall2016/

[4] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY: McGraw-Hill, 2020.

[5] A. Martinez, “Material de ci-0127 bases de datos, universidad de costa rica.”

[6] J. Widom. Relational algebra 1. [Video]. [Online]. Available: https://www.youtube.com/watch?v=tii7xcFilOA&list=PL6hGtHedy2Z4EkgY76QOcueU8lAC4o6c3&index=9

[7] ——. Relational algebra 2. [Video]. [Online]. Available: https://www.youtube.com/watch?v=GkBf2dZAES0&list=PL6hGtHedy2Z4EkgY76QOcueU8lAC4o6c3&index=10

[8] Microsoft. Microsoft technical documentation. [Online]. Available: https://docs.microsoft.com/

[9] E. Malinowski and A. Martínez, *Material de Apoyo de Bases de Datos I*, 1st ed., Universidad de Costa Rica, 2018.