Databases

A Compendium

Editor Andrea Colledan - and.colledan@gmail.com

Nome Cognome - email di altri autori

December 2018

Introduction

This compendium is intended as an aid to the study of databases for the *Basi di Dati* exam (AY 2017-2018) of the Computer Science (8009) curriculum at *Alma Mater Studiorum: University of Bologna.*

The exam topics that are hereby covered include: general notions on databases and DBMSs, the relational model and relational databases, database-specific languages (including the relational algebra, calculi and SQL), database normalization and transactions.

On the other hand, the following are *not* covered: database design, both conceptual and logical (in particular, the ER model is not covered due to its heavy reliance on graphics), the history of databases and information systems, indices, hashes, B^+–Trees and others.

As it should be clear, this compendium is not intended as an exhaustive coverage of the exam topics and the author denies responsibility over any incorrect or missing information, even when not explicitly included in the previous list.

Table of Contents

1 Definitions

Let us start by giving some base definitions and explanations about the terms used in the context of (relational) databases and Database Management Systems (DBMSs).

1.1 Architecture

A DBMS's architecture can be broken down into two main components:

Data Model The data model of a DBMS consists of the constructs used to describe the organization and behavior of data. Through these constructs, types are defined and data are given structure.

Logical Model The logical model of a DBMS describes the way data are stored and linked, independently from their representation in memory. Examples of logical models are the hierarchical, lattice, object, XML and relational ones.

The main distinction between data and logical models is that the former focuses on the individual datum (its type, constraints, structure, etc.), while the latter is more concerned with the relationships between data.

A database can be described in terms of schemata. A schema consists of a description of the abstract structure of a database, without its contents. Given a database, we distinguish between three kinds of schemata:

Internal Schema The internal schema of a database is a description on a low, physical (storage) level of the database. It is a description of the raw structures that store and link the data.

Logical Schema The logical schema of a database is a description of the organization of its data according to the logical model. It is independent from the physical representation of the database.

External Schema An external schema of a database describes a portion of the tables of the database and the relationships within those portions. It is the outermost, user-visible schema and might coincide with the logical one.

Each schema is independent from the ones it is built on. Just as the logical schema is unaffected by changes to the internal schema, the external schemata *should be* unaffected by changes in the logical schema.

1.2 Languages

Languages designed to interact with DBMSs can be split into two categories, based on which part of a database they aim to manipulate:

Data Manipulation Languages (DMLs) DMLs are designed to query databases and to manipulate instances of data inside (instances of) databases.

Data Definition Languages (DDLs) DDLs are used to describe schemata (internal, logical and external) and to operate on a database's structure.

An example of a DDL command (SQL):

```
CREATE TABLE hours (
course CHAR(20),
teacher CHAR(20),
room CHAR(4),
Hour CHAR(5))
```

An example of a DML query (SQL):

```
SELECT lecture, room, floor
FROM rooms, lecture
WHERE name = room AND floor = 'ground'
```

1.3 The Relational Model

While most models (hierarchical, network, etc.) rely on pointers to link data, the relational model uses the values of those data to achieve the same goal.

1.3.1 Relations

The relational model is based on the concept of logical relation.

Definition (Logical Relation). *Let* $D_1, D_2, \ldots, D_n$ *be a number of sets, called domains. A* logical relation *on these domains is any set r so that:*

$$r \subseteq D_1 \times D_2 \times \ldots \times D_n$$

That is, a set of ordered tuples of the form $(d_1, d_2, \ldots, d_n)$*, where* $d_i \in D_i$*.*

These constructs are positional: by swapping any two elements of a tuple we get a different relation, or even an invalid one. By augmenting each element of a tuple with

the name of the domain it belongs to, we get an equally expressive construct that is non-positional.

$$\{d_1, d_2, d_3\} \neq \{d_1, d_3, d_2\}$$

$$\{D_1: d_1,\ D_2: d_2,\ D_3: d_3\} = \{D_1: d_1,\ D_2: d_2,\ D_3: d_3\}$$

This augmented construct lets us represent a relation as a table in which each column header corresponds to a domain and each element e_{ij} of the table corresponds to the value of the i-th tuple with respect to the j-th domain. Any table can be interpreted as a relation as long as:

- All its column headers are different.
- All its rows are different.
- Elements belonging to the same column are homogeneous.

We can now provide some more definitions.

Definition (Relation Schema). *A* relation schema *consists of a relation name R, along with the set of attributes $A_1, A_2, \ldots, A_n$ on which the relation is defined:*

$$R(A_1, A_2, \ldots, A_n)$$

Definition (Database Schema). *A* database schema *consists of a set of relation schemata $R_1, R_2, \ldots, R_n$ along with their attribute sets $X_1, X_2, \ldots, X_n$:*

$$S = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}$$

Definition (Tuple). *A* tuple *t over a set of attributes X consists of a mapping from each attribute $A \in X$ to a value in the domain of A (or NULL). Such value can be referred to as $t[A]$.*

Definition (Relation Instance). *A* relation instance *r on a schema $R(X)$ consists of a set of tuples that follow the schema $R(X)$.*

Definition (Database Instance). *A* database instance *d over a database schema $S = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}$ is a set of relational instances r_i, so that r_i follows $R_i(X_i)$:*

$$d = \{r_1, r_2, \ldots, r_n\}$$

1.3.2 Partial Information

It is possible that relevant data do not match the strict format of a schema, but need to be entered nonetheless into the database. This problem is commonly solved with the introduction of a special value called NULL.

A NULL value can have different meanings. In particular, it can be used to denote:

- An *absent* value
- An *unknown* value
- An *inexistent* value
- An *uninformative* value

1.3.3 Keys

We need a way to uniquely identify tuples inside a table. Of course, by definition, a tuple is uniquely identified by its attributes, but this is useless if our goal is to retrieve such information in the first place. Tuples must be uniquely identifiable by a subset of their attributes.

Definition (Superkey). *Let r be a relation. A* superkey *on r is any subset K of r's attributes so that, for all t_1 and t_2 (distinct) tuples of r, we have:*

$$t_1[K] \neq t_2[K]$$

That is, a superkey uniquely identifies tuples inside a relation. Among all possible superkeys, we are most interested in the ones which are minimal:

Definition (Key). *Let K be a superkey on a relation. K is also a* key *if no proper subset of K is a superkey. That is, if K is minimal.*

Example A superkey in a table of students could be, for example, the subset {*Name, Surname, Birthdate*}, as it is unlikely (although possible) that two different students share all those attributes. A key (minimal superkey) on that same relation could be {*RegistrationID*}.

Primary Keys Usually, one key among all the possible keys of a relation is designated as "preferred" and is actually used to identify tuples. This is called the primary key of a relation. A significant limitation of a primary key is that it cannot contain NULL values.

Foreign Keys Keys play a crucial role in the relational model. In section 1.3 we said that the relational model employs values to link data. When the key of one table appears as a subset of attributes in a different table, we call that key a foreign key in

the second table. Foreign keys realize the actual relationship between the records of different tables.

1.3.4 Constraints

The introduction of NULL values in section 1.3.2 introduces major risks in the form of data inconsistency. That is, when data is so incomplete that it becomes unusable inside a relational database. Furthermore, depending on context, many non-NULL values of the domain of a database may be meaningless or even incorrect (e.g. in a table of exams, grades can only be integer values between 18 and 30).

Integrity Constraints Both these problems are solved through integrity constraints. These constraints are properties that must hold true on every instance of a database. They are expressed through boolean predicates that relate values:

- **Intra-relationally.** That is, within a single table:
  - Domain Constraints
  - Tuple Constraints
- **Inter-relationally.** That is, across multiple tables.

An example of domain constraint would be the one shown in the beginning of this section, in which we pointed out how the grade of an exam is meaningful only when it is an integer between 18 and 30. A tuple constraint on the same domain would be, for example, the requirement that honors be assigned only to exams with a grade of 30 (*"19 with honors"* would be comically meaningless).

An example of a domain constraint definition (SQL):

```
CONSTRAINT grade_chk CHECK (grade >= 18 AND grade <= 30)
```

An example of a tuple constraint definition (SQL):

```
CONSTRAINT honors_chk CHECK (honors = 0 OR grade = 30)
```

Referential integrity constraints On the inter-relational side, we have the problem of referential integrity. In section 1.3.3 we saw how foreign keys, and in particular their equal values across different tables, realize an effective linking of data.

Definition (Referential Integrity Constraint). *A* referential integrity constraint *(or "foreign key" constraint) between the attributes X of a relation* r_1 *and a second relation* r_2 *forces the values concerned by X in* r_1 *to be the values of a key of* r_2.

It is essential for the consistency of a relational database that foreign key constraints be enforced on all relations. Nonetheless, the manipulation of a database instance (e.g. the deletion of a tuple) can create inconsistencies that need to be resolved through *compensatory actions*.

Compensatory actions Let t_1 be a tuple in a relation in which a subset of attributes X constitutes a foreign key to a second tuple t_2 on a different relation. When we try to delete t_2, different actions can be carried out:

- **Restriction:** The deletion is *not* carried out (standard behavior).
- **Cascading:** The referencing tuple t_1 is deleted as well.
- **Nullification:** The attributes of t_1 in X are set to NULL.
- **Default:** The attributes of t_1 in X are reset to some default values.

2 Relational Algebra and Calculi

We have seen in section 1.2 that database languages can be DDLs or DMLs (or both). In this section we show three important theoretical DMLs that serve as bases for real languages. DML query languages can be of two types:

- **Procedural:** when they specify the sequence of *instructions* to be followed in order to obtain the desired result.
- **Declarative:** when they just specify the *properties* of the desired result.

The one procedural language we are going to see is the *relational algebra*, while the declarative languages are the *relational calculi*.

2.1 Relational Algebra

The relational algebra is defined, like any other algebra, through a set of elements – in this case, relations – closed under a number of operations. Relations are sets and as such they are closed under set operations (provided
they follow the same schema). Therefore, our first three operations are:

- **Union:** $r_1 \cup r_2$ contains both the tuples of r_1 and r_2.
- **Intersection:** $r_1 \cap r_2$ contains only the tuples of r_1 which are also tuples of r_2 .
- **Difference:** $r_1 \setminus r_2$ contains only the tuples of r_1 which are *not* tuples of r_2.

Furthermore, the relational algebra provides the following, domain-specific operations:

Renaming The renaming operator (ρ) is used to change the name of an attribute inside a relation. Its syntax is:

$$\rho_{newName \leftarrow oldName}(r)$$

The result is the same relation r, in which the attribute called *oldName* has been renamed to *newName*.

Selection The selection operator (σ) is used to extract specific tuples from a relation. Its syntax is:

$$\sigma_{predicate}(r)$$

Where *predicate* is a boolean expressions over the attributes of r. The result is a relation with the same schema of r that contains only those tuples of r which satisfy *predicate*.

Projection The projection operator (π) is used to extract specific attributes from a relation. Its syntax is:

$$\pi_{attributeList}(r)$$

The result is a relation that contains all the tuples in r, only restricted to the attributes in *attributeList*.

Note that by compounding a selection (vertical extraction) with a projection (horizontal extraction) we can extract individual pieces of information from a table, with arbitrary granularity. This compound operation has the same semantics of the SELECT operation in SQL (see section 3.2.1).

Join The join operations (usually denoted with ⋈) are used to correlate (that is, join) different relations. The basic syntax of a join operation is:

$$r_1 \bowtie r_2$$

Different variants of join operator exist, with different semantics. The next section is devoted to these variants.

2.1.1 The Join Operator

The join operator is used to correlate different relations and as such it is the operator that actually realizes the relational aspect of the language.

Natural join The natural join operator (⋈) is the simplest form of join operator. Its syntax is:

$$r_1 \bowtie r_2$$

The result is a relation r whose schema is the union of r_1 and r_2's schemata and whose tuples constitute the following set:

$$\{t \text{ on } X_1 \cup X_2 | \exists t_1 \in r_1 . \exists t_2 \in r_2 . t[X_1] = t_1 \wedge t[X_2] = t_2\}$$

Where X_1 and X_2 are the attribute sets of r_1 and r_2, respectively. In other words, if the two relations have no attributes in common, every concatenation of $t_1 \in r_1$ and $t_2 \in r_2$ is found in the result, which coincides with the Cartesian product of the two

relations. On the other hand, if the two relations *do* share one or more attributes, the result contains the concatenation of only those tuples which share the same values over those attributes.

In general, the result of a natural join is smaller than $|r_1| \times |r_2|$, as tuples that do not match over the two relations are discarded. When all tuples appear in the final result, the join operation is said to be a *full join*, whereas if no tuple appears in the final result (no tuple matches over the relations) the join operation is said to be an *empty join*.

Outer join The outer join operator (⟕) behaves just like the natural join, with the difference that *all* the tuples from the two relations involved appear in the final result. When a tuple does not have a match in the other table, its missing attributes are set to NULL.

Depending on which side of the outer join the partial, NULL-padded tuples are allowed from, we can distinguish between:

- Full outer join (partial tuples from both relations: ⟗)
- Left outer join (partial tuples from the the first relation only: ⟕)
- Right outer join (partial tuples from the second relation only: ⟖)

Theta-join The theta-join operator is actually a compound operator. Since selections after a natural join (or Cartesian product) are common and have very specific semantics (selecting tuples over tables that are related in a *specific* way), an operator is defined that expresses these semantics, i.e. the theta-join. Formally:

$$r_1 \bowtie_{predicate} r_2 = \sigma_{predicate}(r_1 \bowtie r_2)$$

Where *predicate* is usually a conjunction (AND) of comparisons across tables.

Equi-join The equi-join is a particular case of the theta-join, in which the only comparisons employed in *predicate* are equivalences. The equi-join allows the joining of relations over attributes with different names, an operation that would usually require a renaming and a natural join. Formally:

$$r_1 \bowtie_{att_1 = att_2} r_2 = r_1 \bowtie \rho_{att_1 \leftarrow att_2}(r_2)$$

Where att_1 and att_2 are attributes of r_1 and r_2, respectively. The equi-join is by far the most employed kind of theta-join.

2.1.2 Null Values

It is important to note that atomic predicates in the relational algebra (in selections, equi-joins and so on) are only satisfied by non-NULL values, unless explicitly specified otherwise. For example, on a table of people, we have that:

$$\sigma_{Age > 30}(People) \cup \sigma_{Age \leq 30}(People) \neq People$$

In particular, the two united selections do not include people whose *Age* is NULL. To deal with possibly-NULL values, two ad-hoc atomic predicates are provided: *IS NULL* and *IS NOT NULL*. We have that:

$$\sigma_{Age > 30}(People) \cup \sigma_{Age \leq 30}(People) \cup \sigma_{Age\ IS\ NULL}(People) = People$$

2.2 Relational Calculi

Relational calculi are a family of DMLs based on first order logic. Two different definitions exist: the *Domain Relational Calculus* and the *Tuple Relational Calculus with Range Declarations*.

2.2.1 Domain Relational Calculus

In the DRC, a query is an expression in the form:

$$\{A_1 : x_1, \ldots, A_n : x_n \mid f\}$$

Where f is a FOL formula with comparisons, $A_1, \ldots, A_n$ are different attributes, possibly across several tables or databases, and $x_1, \ldots, x_n$ are different variables. $A_1 : x_1, \ldots, A_n : x_n$ defines the *target list* of the expression. The result of such an expression is a relation over $A_1, \ldots, A_n$ which contains
those tuples whose values $x_1, \ldots, x_n$ satisfy f.

Examples Here are some examples of queries on a database of employees and supervisors, first described using natural language, then using the relational algebra and lastly using the DRC.

- Employee number, name and salary of those employees earning more than 400$/week.

$$\sigma_{Salary > 400}(\text{Employee})$$

$$\{\text{Number} : n, \text{Name} : m, \text{Salary} : s \mid \text{Employee}(\text{Number} : n, \text{Name} : m, \text{Salary} : s) \wedge s > 400\}$$

- All the names of the employees.

$$\pi_{Name}(\text{Employee})$$

$$\{\text{Name} : m \mid \exists n.\exists s.\text{Employee}(\text{Number} : n, \text{Name} : m, \text{Salary} : s)\}$$

- The numbers of those supervisors with at least one employee earning more than 500$/week.

$$\pi_{Chief}(\text{Supervisor} \bowtie_{Employee=Number} \sigma_{Salary > 500}(\text{Employee}))$$

$$\{\text{Chief} : c \mid \exists e.(\text{Supervisor}(\text{Chief} : c, \text{Employee} : e) \wedge \exists n.\exists s.\text{Employee}(\text{Number} : e, \text{Name} : n, \text{Salary} : s) \wedge s > 500)\}$$

Pros and cons The fact that the DRC is declarative is a huge advantage over other procedural languages (the result is defined in terms of desired properties and not of complex procedures). We can make two observation on the complexity of DRC queries. One one hand, we see how the complexity of a DRC expression grows slowly with respect to the complexity of the query (and of the corresponding algebraic expression). On the other hand, this slow growth rate comes at a price, as even the simplest query (a simple selection) corresponds to a very verbose and possibly obscure DRC expression.

Also note that many valid expressions of the DRC, such as $\{A: x \mid \neg R(A: x)\}$, are meaningless, as they are domain dependent (in the example, the result depends on the extension of domain A). Note that if we only consider domain independent expressions, the DRC and the relational algebra are equivalent.

2.2.2 Tuple Relational Calculus with Range Declaration

With the TRC our main goal is to overcome the flaws of the DRC. That is, the unnecessary verbosity and the presence of domain dependent expressions. In the TRC, a query is an expression in the form:

$$\{target \mid range \mid formula\}$$

Where target contains elements of the form $X: y.Z$ (we are targeting the attributes of y in Z under the names in X), *range* specifies where each variable in *target* is to be taken from (e.g. y(Employee)) and *formula* is a condition on the values of the targets.

Examples We now see how the examples of the previous section become more legible in the TRC.

- Employee number, name and salary of those employees earning more than 400$/week.

$$\{\text{Number} : n, \text{Name} : m, \text{Salary} : s \mid \text{Employee}(\text{Number} : n, \text{Name} : m, \text{Salary} : s) \wedge s > 400\}$$

$$\{i.* \mid i(\text{Employee}) \mid i.\text{Salary} > 400\}$$

- All the names of the employees.

$$\{\text{Name} : m \mid \exists n.\exists s.\text{Employee}(\text{Number} : n, \text{Name} : m, \text{Salary} : s)\}$$

$$\{i.\text{Name} \mid i(\text{Employee}) \mid \}$$

- The numbers of those supervisors with at least one employee earning more than 500$/week.

$$\{\text{Chief} : c \mid \exists e.(\text{Supervisor}(\text{Chief} : c, \text{Employee} : e) \wedge \exists n.\exists s.\text{Employee}(\text{Number} : e, \text{Name} : n, \text{Salary} : s) \wedge s > 500)\}$$

$$\{i.\text{Chief} \mid i(\text{Supervisor}), e(\text{Employee}) \mid i.\text{Employee} = e.\text{Number} \wedge e.\text{Salary} > 500\}$$

Pros and Cons The TRC retains the advantages of a declarative DML and is furthermore leaner than the DRC. It is also free from domain dependent expressions, as each variable is assigned a specific, concrete range on the database. However, the TRC cannot express some significant queries, such as unions.

2.3 Limitations of Relational Algebra and Calculi

The three languages we presented all suffer from some form of limitation. Namely, it impossible in both the algebra and the calculi to:

- Use the extracted values as parameters to further query a database.
- Perform arithmetic on a tuple (summation, conversion, etc.).
- Perform arithmetic over multiple tuples (averages, summations, etc.).
- Express recursive queries, such as the transitive closure.

These limitations are overcome in real languages (e.g. SQL) through explicit constructs (note that SQL is a hybrid between a procedural and a declarative language).

3 SQL

SQL (originally an acronym for *Structured Query Language*) is a hybrid declarative language that implements both a DDL and a DML. Although in practice every SQL-based DBMS defines its particular SQL grammar, the core components are basically identical in every implementation.

3.1 SQL as a DDL

The first SQL command run to create a database is:

```
CREATE DATABASE db_name
```

Which opens a database named *db_name*. To define a schema for such a database (to describe views, constraints, domains, etc.) we can use the following command:

```
CREATE SCHEMA schema_name
AUTHORIZATION 'username'
```

Which creates a schema named *schema_name,* belonging to the user *username*.

Table creation The first real step in creating a database comes with the creation of a relation (table):

```
CREATE TABLE table_name(
     attr_1 type_1 extra_1,
     attr_2 type_2 extra_2,
     …
     attr_n type_n extra_n,
     FOREIGN KEY (key)
     REFERENCES table (key),
     other…
)
```

We will now proceed to analyze the different parts of this template in greater detail.

3.1.1 Attribute Data Types

SQL, like most languages, provides some basic data types:

- **CHAR(n):** a character or string of fixed length.
- **STRING:** a string of variable length.
- **NUMERIC:** an integer or floating-point number.
- **DATE**
- **TIME**
- **INTERVAL**
- **BOOLEAN**
- **BLOB:** *Binary Large OBject,* for storing large binary data.
- **CLOB:** *Character Large OBject,* for storing large textual data.

Furthermore, SQL allows to define custom data types through the *domain* constructor:

```
CREATE DOMAIN dom_name
AS base_type DEFAULT def_value
CHECK constraint
```

Which creates a new domain (type) of name *dom_name* starting from an already existing type *base_type*. The new type must obey the constraints defined through the CHECK statement.

3.1.2 Table Constraints

The fields labeled extra in the table creation template can contain table constraints, that is, special properties that always hold true on the table. These can be:

- **NOT NULL:** after an attribute, it specifies that such attribute can never hold a NULL value.
- **PRIMARY KEY:** after an attribute, it specifies that such attribute serves as a primary key for its table (entails NOT NULL).
- **UNIQUE(attrList):** stand-alone or after an attribute (in this case *attrList* is omitted), it specifies that the attributes in *attrList* can be keys of the relation.
- **CHECK:** like we saw before. Enforces a constraint on the table.

Referential integrity An important kind of constraint is the one defined by the FOREIGN KEY ... REFERENCES construct, which creates a referential integrity constraint (on a single attribute or on multiple attributes). When this construct is employed, we can define *referential triggered actions*:

```
ON <DELETE | UPDATE>
<NO ACTION | CASCADE | SET NULL | SET DEFAULT>
```

The context and semantics of these actions are exactly those of the compensatory actions described in section 1.3.4.

3.1.3 ALTER and DROP

So far we have only seen CREATE commands. For each of the schema entities described in this section, an ALTER and a DROP command also exist to redefine and delete that entity, respectively.

Domains As for domains, the ALTER DOMAIN command must be followed by one among SET DEFAULT (changes default value), DROP DEFAULT (resets such a value), ADD CONSTRAINT (defines a new named CHECK constraint on the domain) and DROP CONSTRAINT (removes such constraint).

DROP DOMAIN, on the other hand, simply removes a user-defined domain from the system.

Tables As for tables, the ALTER TABLE command is useful when followed by one among ALTER COLUMN (changes the properties of a column), ADD COLUMN (defines a new column), DROP COLUMN (removes a column), ADD CONSTRAINT (defines a new named CHECK constraint on the table) and DROP CONSTRAINT (removes a constraint).

As with domains, a DROP TABLE instruction entirely removes a table from a database (both the schema and the data).

3.2 SQL as a DML

The main instructions of SQL's DML component are SELECT, INSERT, UPDATE and DELETE. Among these, SELECT is by far the most important.

3.2.1 SELECT

The SELECT statement is the most expressive statement of SQL's DML. It is used to retrieve data from tables and follows this basic syntax:

```
SELECT attributes
FROM tables
WHERE condition
```

Such an instructions behaves like a projected selection in the relational algebra, that is, it returns the tuples from *tables* (more on why "*tables*" and not "*table*" later) that satisfy *condition*, restricted to the attributes specified in *attributes* ('∗' can be used to

denote all the attributes of the schema). The same query in the relational algebra would look like:

$$\pi_{attributes}(\sigma_{condition}(\text{tables}))$$

The SELECT statement also supports in-line renaming of tables and attributes. Here is an example:

```
SELECT p.first_name AS name
FROM people p
WHERE p.age = 38
```

This returns all the names of people who are 38 years old. The SELECT statement also supports arithmetic manipulation of the results and pattern matching:

```
SELECT p.salary/4 AS weekly_salary
FROM people p
WHERE p.name LIKE 'J_m%'
```

This returns a table with the weekly salaries of all people whose name begins with 'J' and contains an 'm' as third character. Lastly, the WHERE clause can include IS (NOT) NULL checks.

Joins The FROM clause lets us specify as many tables as we desire. If no WHERE clause is specified after that, the resulting table on which the query is run coincides with the Cartesian product of the tables. If a WHERE clause is specified, the resulting table is the output of an (implicit) theta-join which uses the conditions of the WHERE clause.

```
SELECT f.child, father, mother
FROM fatherhood f, motherhood m
WHERE f.child = m.child
```

Returns a table in which children are listed along with their father and mother. Notice how this query requires that the two tables *fatherhood* and *motherhood* be joined on the condition that the two parents have the same child. Also note how this is not only an equi-join, but also a natural join (as the attribute name is the same in the two tables). Joins can also be explicitly expressed:

```
SELECT fatherhood.child, father, mother
FROM fatherhood JOIN motherhood
ON fatherhood.child = motherhood.child
```

```
SELECT child, father, mother
FROM fatherhood NATURAL JOIN motherhood
```

Both return the same results of the first query, provided both *motherhood* and *fatherhood* posses an attribute named *child* (note that the ON clause is specific to joins). Explicit outer joins (left, right and full) are also available, with the same semantics of the corresponding operators in the relational algebra.

Sorting The result of a SELECT statement is a list of tuples, which can be sorted. A statement such as:

```
SELECT child, father, mother
FROM fatherhood NATURAL JOIN motherhood
ORDER BY child <ASC | DESC>
```

Sorts the output of the previous query according to the values of *child*, in an ascending (ASC, default) or descending (DESC) fashion.

Union The results of several selections can be united like sets (no duplicate tuples allowed) or multisets (duplicate tuples are preserved) with UNION and UNION ALL:

```
SELECT …
  UNION [ALL]
SELECT …
```

Note that when an union between relations with different schemata is attempted, the schema of the first selection result overrides the schema of all the following results.

Difference Difference between selections is also allowed:

```
SELECT name                                SELECT name
FROM employee                              FROM employee
WHERE age > 30                  =          WHERE age > 30
  EXCEPT                                   
```

```
SELECT name                                    AND age <= 45
FROM employee
WHERE age > 45
```

Intersection Lastly, the intersection between relations is also allowed:

```
SELECT name
FROM employee
  INTERSECT
SELECT surname AS name
FROM employee
```

Returns the relation of names which are also surnames among employees.

Nested queries When we described the relational algebra and calculi we saw how a significant limitation of both formalisms was that data extracted from a database could not be used to query more data. This is not the case with SQL, as nested (sub)queries are allowed:

```
SELECT age
FROM people
WHERE name = (
          SELECT father
          FROM fatherhood
          WHERE child = 'Frank')
```

Returns the age of Frank's father, as the sub-query's result is the name of Frank's father. Nested queries are particularly useful when used in conjunction with the ANY, ALL, IN and EXISTS constructs.

```
SELECT age
FROM people
WHERE name = ANY(
          SELECT child
          FROM fatherhood
          WHERE father = 'Adam')
```

Returns the ages of all of Adam's children (an age is returned if the name of the person appears within the results of the subquery). Note that in this case the same query in which "= ANY" is replaced with "IN" is equivalent.

As we have already seen, “Attribute operator ANY(...)” is true if the operation is true over *Attribute* and at least one element of the ANY clause. “Attribute operator ALL(...)” is true if the operation is true over *Attribute* and all of the elements inside the ALL clause. “Attribute IN(...)” is true if *Attribute* appears in the body of the IN clause. Lastly, “EXISTS(...)” is true if its body is non-empty.

Subquery scope Scope rules are traditional: each subquery has access to its variables and to its superqueries’ variables and is forbidden access to its subqueries’ variables. Other important aspects of subqueries is that each subquery is run once for every tuple in its superquery (this approach is more procedural than declarative) and that set operations are *not* available in subqueries.

```
SELECT name
FROM employee e
WHERE NOT EXISTS(
          SELECT *
          FROM employee
          WHERE surname = e.name)
```

Aggregate functions The immediate body of a SELECT statement (that is, what we called *attributes* at the beginning of this section) can contain functions over the tuples that have been extracted. Such operations include:

- **COUNT:** counts the number of tuples in the result.
- **SUM:** calculates a summation over the resulting tuples.
- **AVG:** calculates an average value from the resulting tuples.
- **MAX:** computes a maximum value from the resulting tuples.
- **MIN:** computes a minimum value from the resulting tuples.

The syntax of an aggregate function is:

```
SELECT aggregator ([DISTINCT] attributes)
FROM …
```

Note that aggregate functions disregard NULL values. When using aggregations, it is sometimes useful to group tuples.

Grouping By appending the “GROUP BY *attributes*” clause to a selection, we can group the tuples of a table over *attributes* and then perform aggregations on the remaining attributes of each group. The additional construct HAVING lets us define conditions over aggregations which determine which tuples make it to the final result.

```
SELECT supplier, AVG(unitPrice)
FROM product
GROUP BY supplier
HAVING COUNT(DISTINCT productId) > 2
```

This last examples returns a table in which every supplier is listed along with the average price of the products he supplies, provided he supplies at least two different products.

At this point, we can exhibit the extended syntax of a SELECT statement:

```
SELECT attributes_1 and expressions
FROM tables with joins
[WHERE condition]
[GROUP BY attributes_2]
[HAVING aggregate condition]
[ORDER BY attribute]
```

3.2.2 INSERT

The INSERT statement lets us add tuples to a table. Its syntax is:

```
INSERT INTO table [(attributes)]
VALUES (vals)
```

This instruction inserts into *table* a tuple which contains values *vals* over *attributes*. Note that the VALUES clause can be replaced by a SELECT subquery, whose results will be added to *table*. If *attributes* is omitted, all of the attributes from *table* are assumed, in the order they appear in the schema (insertion is, in this case, positional).

3.2.3 UPDATE

The UPDATE statement allows us to edit already existing tuples in a table. Its syntax is:

```
UPDATE table
SET attribute =
<expression | NULL | SELECT … | DEFAULT>
[WHERE condition]
```

Such a statement sets *attribute* to the value after '=' in all the tuples of *table* that satisfy *condition*.

3.2.4 DELETE

Lastly, DELETE allows us to delete tuples from a table. Its syntax is:

```
DELETE FROM table
[WHERE condition]
```

Such a statement deletes all tuples of *table* that satisfy *condition*. Care must be taken, as a DELETE statement without a WHERE clause is interpreted as having an always true *condition* and deletes all of *table*'s tuples.

DELETE (and also UPDATE) may cause cascading deletions (or updates) if the policy of the database specifies CASCADE as a rule for DELETE (or UPDATE).

3.3 Advanced SQL

The following constructs are not implemented in all SQL DBMSs, but are still relevant.

3.3.1 CHECK and ASSERTION

We have already seen how a CHECK statement behaves. In practice, CHECK statements on their own are limited to individual tuples and do not support nested selections. ASSERTION checks, on the other hand, allow the definition of constraints over multiple tables (and even entire databases). Assertions are seldom supported by real DBMSs.

```
CREATE ASSERTION atLeastOneEmployee
CHECK (SELECT COUNT(*) FROM employee) > 0
```

3.3.2 Views

SQL allows the definition of views (derived tables) through the construct VIEW. Its syntax is:

```
CREATE VIEW name [(attributes)] AS
SELECT …
[WITH [LOCAL | CASCADED] CHECK OPTION]
```

This statement creates a view that exposes *attributes* as extracted by the SELECT statement in the second line. The last line may be used to enforce that all updates/insertions to the view respect its constraints (as may be defined inside the SELECT's WHERE). LOCAL and CASCADED allow to choose whether to consider only the local view's constraints or all the constraints of the base tables and views the local view is built on.

Note that queries on a view are automatically expanded by the DBMS into queries on the base tables.

3.3.3 Recursive Queries

A particularly useful type of view is the one defined recursively using the WITH RECURSIVE statement. Assume that we have a *fatherhood* table and that we want to define a view of *ancestors* on that table. Formally, we can capture the concept of ancestry by writing:

$$ancestors(a, d) \Leftarrow fatherhood(a, d).$$

$$ancestors(a, d) \Leftarrow ancestors(a, i)\ ,\ fatherhood(i, d).$$

The same concept is expressed by the following SQL statement:

```
WITH RECURSIVE ancestors(ancestor, descendant) AS
(    SELECT father, child
     FROM fatherhood
UNION ALL
     SELECT ancestor, child
     FROM ancestors, fatherhood
     WHERE descendant = father
)
```

3.3.4 Scalar Functions

SQL provides some built-in scalar functions to retrieve and manipulate values. These functions include:

- **current_date:** returns the current date.
- **extract:** takes "*component FROM date*" as an argument and returns the value of *date* over *component*.
- **char_length:** returns how many characters long a string is.

- **lower:** returns the lowercase version of a string.

3.3.5 COALESCE

The COALESCE function takes a number of attributes or values as input and returns the first of those inputs which is not NULL. It can be used to automatically make up for missing data or to define custom NULL values:

```
SELECT name,
COALESCE (mobile, phone_number, 'Unavailable')
AS contact
FROM participants
```

3.3.6 NULLIF

The NULLIF function takes an attribute and a literal as input and returns NULL if the attribute equals the literal, otherwise it returns the value of the attribute:

```
SELECT name, NULLIF(contact, 'Unavailable')
FROM participants
```

3.3.7 CASE

The CASE construct allows to define if-then-else (or switch) -like structures. Using a CASE construct, a selection's entries can be built differently depending on the values they are computed from.

```
SELECT name,
(CASE paymentFrequency
     WHEN 'Weekly' THEN 4*Salary
     WHEN 'Monthly' THEN Salary
     ELSE NULL
END) AS monthlySalary
FROM employee
```

3.4 Privileges

On the non-functional side, SQL offers *privileges* as a way to implement security policies over a database. Except for the default admin (e.g. *_System*), each user has a limited set of privileges.

Privilege A privilege consists of a quintuple p of the form:

$$p = \{granting, subject, operation, resource, propagable\}$$

That is, user *granting* has allowed user *subject* to perform *operation* on *resource* and may or may not (depending on *propagable*) have allowed them to extend their privilege to other users.

The *operations* of the previous definition include, alongside with SQL's select, insert, delete and update, the *reference* operation (to define referential constraints) and the *usage* operation (to use constructs such as data types). The syntax for granting privileges is as follows:

```
GRANT <privileges | ALL PRIVILEGES>
ON resource
TO users
[WITH GRANT OPTION]
```

Such statement grants the privileges in *privileges* (or all privileges) on *resource* to the users in *users*. If WITH GRANT OPTION is used, the users in *users* can propagate their *privileges* to other users, with the same syntax. The syntax to revoke privileges, on the other hand, is:

```
REVOKE privileges
ON resource
FROM users
[RESTRICT | CASCADE]
```

In this case, the same user who granted the privileges revokes them. If RESTRICT is specified, then only the users in *users* are revoked their privileges. On the other hand, if CASCADE is specified, then all the users who received their privileges from the users in *users* lose their privileges as well.

Note that just because a user is not allowed to access a table, it does not mean that such user cannot access views derived from that table. In fact, views are frequently used as a means to restrict table access to a limited set of meaningful attributes.

RBAC SQL-3 was extended to support the Role-Based Access Control (RBAC) protocol. In RBAC, a *role* is a container of different privileges, granted through the GRANT command. Each user may be assigned one or several roles, whose privileges are added to the individual user's privileges to determine their total privilege.

```
CREATE ROLE name

SET ROLE name

GRANT name TO user
```

These three commands create a role named *name* and assign it to the current user and to *user*, respectively.

3.5 Triggers

When we examined referential compensatory actions, we saw that SQL offered constructs to define the desired behavior of the DBMS in case the reference was broken or altered. Triggers (*active rules* in a SQL-independent context) are a generalization of this concept, as they allow to define actions to be undertaken by the DBMS (we speak of an *active database*) on almost arbitrary conditions.

A trigger is defined by an Event-Condition-Action (ECA) sequence. Informally, when an event occurs a condition is checked. If such condition is true, then some actions are undertaken. Note that triggers *are part of the database*. As such, they are shared among all the applications which access it, they reduce the problem of impedance mismatch and they are not bypassable.

Trigger definition syntax (Oracle):

```
CREATE TRIGGER name
mode event {, event}
ON table
[[REFERENCING reference
FOR EACH granularity
[WHEN sql_predicate]]
action
```

Let's break this syntax down: *mode* defines whether *action* (Usually a PL/SQL or SQL block) is to be undertaken *before* or *after* the operation specified in *event* (which can be *insert*, *delete*, *update*). In *reference* we can have variable declarations of the usual form (*attribute* AS *variable*). FOR EACH lets us decide which *granularity* we execute our actions with. That is, whether the trigger is fired once for every *row* of the triggering query, or just once for the whole *statement*.

Extensions Some (usually unsupported) extensions of database triggers include the possibility to define time-triggered and custom events, the INSTEAD OF construct (which entirely substitutes the triggering action with a different one) and the definition of priorities.

3.6 Transactions

SQL supports transactions (see next section). Each transaction is started by a START TRANSACTION statement (optional) and is ended by a COMMIT or ROLLBACK statement, depending on whether the changes are to be kept or discarded.

4 Transactions

A transaction consists of a sequence of database manipulations treated as a logical unit of database processing. Transactions respect the ACID properties:

Atomicity A transaction is either performed fully or not at all.

Consistency If a transaction is executed completely, the database moves between two consistent states. If consistency violations occur during the transaction and they are not resolved by the end of the transaction, such transaction is aborted entirely and the database is not affected.

Isolation Even when executed concurrently, transactions do not interfere with each other.

Durability Once a transaction is confirmed (committed), its changes persist, even in case of hardware or software faults.

Of course, a transaction (or better, its operations) can be reduced to a sequence of simple read/write operations on the data handled by the DBMS. Users see their transactions carried out simultaneously, but in fact all the basic operations inside their transactions are (transparently) executed in an interleaved fashion.

Definition (Scheduling). *Let* $T_1, T_2, \ldots, T_n$ *be transactions, each with its operations:* $T_i = \{O_i^1, O_i^2, \ldots, O_i^k\}$. *A scheduling (or history)* S *of* $T_1, T_2, \ldots, T_n$ *is a total ordering of the operations of all* T_i. *That means that* S *is a sequence of operations such that for all* i*:*

$$O_i^p \preccurlyeq O_i^q \text{ in } T_i \Rightarrow O_i^p \preccurlyeq O_i^q \text{ in } S$$

We now analyze the threats posed to the ACID properties in two delicate scenarios: concurrency and crash recovery. Note that while the consistency part of ACID depends in part on a correct implementation of the transaction by the user, the other properties depend entirely on the actual implementation of the DBMS (and concern the low-level mechanisms of a database).

4.1 Concurrency Control

When transactions are executed simultaneously, anomalies are likely to occur. Different transactions may interfere with each other (breaks isolation) or the failure of a transaction may undo the changes brought by other, successful transactions

(breaks durability). Interference is our main concern. To understand the next section, it is crucial to understand that every transaction must explicitly terminate, either with a commit operation (the transaction is successful) or with an abort operation (the transaction failed, either due to an exception or a crash). We start with the following definitions:

Definition (Completeness). *A scheduling S of transactions is said to be* complete *if every operation of every transaction (including every commit and abort) is included in S.*

Definition (Seriality). *A scheduling S of transactions* $T_1, T_2, \ldots, T_n$ *is said to be* serial *if for every transaction* T_i*, all the operations in* T_i *are executed consecutively (only one transaction is active at a time).*

Definition (Serializability). *A scheduling S of transactions is said to be* serializable *if its final state is the same as that of a serial scheduling S' in which all transactions commit.*

4.1.1 Conflicts and Anomalies

We now give a more formal definition of the source of most of our problems:

Definition (Conflict). *Two operations* $O_i^p(X)$ *and* $O_j^q(Y)$ *in a scheduling S generate a* conflict *if, at the same time:*

- *They belong to different transactions* $(i \neq j)$.
- *They operate on the same item* $(X = Y)$.
- *At least one between* O_i^p *and* O_j^q *is a write operation.*

Given this definition, we find that three different kind of conflicts can occur: *read-write, write-read* and *write-write.*

Read-write conflict This is the case when a transactions reads a value that is later updated (written) by a different, successful transaction. The first transaction ends up operating on old and incorrect values. The resulting anomaly is know as *non-repeatable read.*

Write-read conflict This is the case when a transactions writes a value that is later read by a second transaction. If the former aborts, the latter ends up operating on an incorrectly updated (and possibly harmful) value. The resulting anomaly is known as *dirty read.*

Write-write conflict This is the case when the same value is written by two different, concurrent transactions. Depending on the position of each write in the scheduling, the transactions may lead to two different database states. The resulting anomaly is known as *lost update*.

A fourth anomaly can be described, which does not depend on the previous kinds of conflict:

Phantom This is the case when a transaction queries a table while a different transaction inserts records into the same table. Every time the first transaction performs the same query, different results are obtained, even though the two transactions handle different objects.

One could object that a dirty read scenario is easily solvable by propagating the abort operation from the offending transaction to all those transactions that used values affected by it. However, this would require that all the transactions that have already terminated by the time the abort takes place be undone, and the durability constraint would be broken (successful transactions may get undone). This solution is therefore unfeasible and such scenarios must be avoided in the first place.

It is evident that not *all* the possible schedulings of n transactions can be allowed. Some checks have to be carried out on the schedulings and some actions must be undertaken when a transaction aborts.

4.1.2 Transaction Implementation

We start by presenting SQL's approach to transactions (and their interaction). The main means of concurrency control in a SQL DBMS is that of *locks*. Locks define data-level constraints that control the access to the individual objects of a database. Locks can be of two types:

- **Exclusive Locks:** if a transaction obtains an exclusive lock on an object, then no other transaction can obtain locks of any kind on that object.
- **Shared Locks:** if a transaction obtains a shared lock on an object, then other transactions can obtain a shared lock on that object, but not an exclusive one.

Locks are handled by the lock manager of a DBMS, through a lock table. That said, each SQL transaction is assigned (at least) the following three features:

- **Access Mode:** determines whether the transaction only retrieves data (READ ONLY) or manipulates the database as well (READ WRITE).
- **Diagnostic Area Size:** determines how many feedback conditions are held simultaneously.

- **Isolation Level:** determines how transactions interact with each other when accessing the database concurrently.

The syntax to define a transaction, along with its properties, is:

```
EXEC SQL SET TRANSACTION
<READ ONLY | READ WRITE>
DIAGNOSTIC SIZE number
ISOLATION LEVEL <SERIALIZABLE
|REPEATABLE READ
|READ COMMITTED
|READ UNCOMMITTED>
```

- **SERIALIZABLE:** when reading or updating any object of the database, the transaction has to hold a lock on that object until its termination. Table-level locks are included (to prevent phantoms).
- **REPEATABLE READ:** when reading or updating any object of the database, the transaction has to hold a lock on that object until its termination. Table-level locks are *not* included.
- **READ COMMITTED:** when updating an object, the transaction requests and holds an exclusive lock. When reading from an object, the transaction asks for a shared lock that is released right after the read operation.
- **READ UNCOMMITTED:** when updating an object, the transaction requests and holds an exclusive lock, while no locks are required for reading.

	Lost Update	Dirty Read	Non-repeatable Read	Phantom
READ UNCOMMITTED	**Possible**	**Possible**	**Possible**	**Possible**
READ COMMITTED	**Possible**	**Prevented**	**Possible**	**Possible**
REPEATABLE READ	**Prevented**	**Prevented**	**Prevented**	**Possible**
SERIALIZABLE	**Prevented**	**Prevented**	**Prevented**	**Prevented**

Isolation levels and the anomalies they prevent.

4.1.3 Alternatives

Other, similar approaches can be employed to solve concurrency anomalies. These include:

Conflict serialization A serializable scheduling is forced through data locking protocols.

Conflict detection All transactions are run concurrently and checks to detect possible conflicts are only performed at commit time. A transaction is split into three phases:

- **Read Phase:** a transaction can read committed values from the database. In this phase, updates are only carried out on *local copies* of the objects involved.
- **Validation Phase:** when the transaction requests that its changes be committed, the database checks if any conflicts or anomalies have arisen. If so, the transaction is automatically aborted and restarted.
- **Write Phase:** if the transaction passes the validation phase (there are no conflicts), then its changes are committed to the database.

Timestamps Timestamps are assigned to transactions that have either written or read objects. Timestamps are later compared to determine in which order the transactions should be run.

Let T_i and T_j be transactions and O_i^p and O_j^q two conflicting operations from T_i and T_j, respectively. If $O_i^p \preccurlyeq O_j^q$ in a scheduling (O_i^p is to be run before O_j^q), we identify two possible scenarios:

- If $TS(T_i) < TS(T_j)$ then the operations are run as scheduled.
- If $TS(T_i) > TS(T_j)$ it means that the ordering is incorrect. T_i is aborted and run again with a greater timestamp.

Strict Two-Phase Locking Writes must use exclusive locks and reads must use shared locks. Locks are released at commit time. Note that with this approach, transaction that handle the same objects are always run serially.

4.1.4 Deadlock Prevention

When locks are employed, the risk of deadlock scenarios arises. DBMSs usually employ a priority system to prevent these situations.

Priority Let T_i and T_j be transactions. Each is assigned a timestamp (e.g. $TS(T_i)$) which acts as a priority (the lower the timestamp, the older the transaction, the higher the priority). When T_i asks for a lock held by T_j, we can define two possible policies:

- **Wait-die:** according to this policy, if $TS(T_i) < TS(T_j)$ (T_i is older than T_j), then T_i is allowed to wait for the release of T_j's lock. Otherwise, T_i is aborted and run again after a delay, but with the same priority (older transactions wait, younger transactions are aborted: a non-preemptive approach).
- **Wound-wait:** according to this policy, if $TS(T_i) < TS(T_j)$, then T_j is aborted to give way to T_i. T_j is run again with the same priority after a short delay. Otherwise, if $TS(T_i) > TS(T_j)$, T_i waits for T_j's lock to be released (older transactions are executed immediately, younger transactions must wait: a preemptive approach).

Detection If deadlocks are expected to be very rare, a simpler (but more expensive) detection policy might be adopted. The DBMS could store a wait-for graph and use it to identify deadlock cycles and decide which transactions need to be aborted to release the deadlock. Alternatively, the DBMS could assign each transaction an inactivity timeout, after which the transaction is considered deadlocked and gets aborted.

4.2 Crash Recovery

A database's *recovery manager* must guarantee that the database will remain consistent after system crashes. We know that transactions can temporarily bring the database into an inconsistent state, and that before committing the state must be consistent again. But what if the system crashes halfway through the transaction? Most recovery protocols are based on logging.

4.2.1 Logging

A log is a collection of records that keep track of the transaction operations that actually affect the database. All log records contain the following data:

- **LSN:** Log Sequence Number. It is an automatically generated, monotonic number that uniquely identifies the log record.
- **Type:** The type of the record.
- **transID:** The ID of the transaction that carried out the operation.
- **prevLSN:** The LSN of the previous operation in the same transaction.

Memory On the physical level, logs are append-only files in secondary memory. The last part of a log (the *log tail*) is usually kept as a buffer in the main memory for a more efficient access.

An example of log syntax:

```
…
[start_transaction, T1]
[read_item, T1, X]
[start_transaction, T2]
[write_item, T1, X, old_value, new_value]
[read_item, T2, Y]
[abort, T2]
[commit, T1]
…
```

The records in the previous example should be self-explanatory. Note that special log records may be present as well, such as end records (when after a commit/abort operation some finalization must take place) or undone records (when actions are undone after a transaction aborts). The latter are know as Compensation Log Records (CLRs).

4.2.2 Write-Ahead Logging

The Write-Ahead Logging (WAL) protocol is used when in-place updates are performed. The changes brought by a transaction are first recorded to the log, only to be written permanently in memory right before a commit takes place. The WAL protocol requires that:

- No item *before-image* be overwritten by its *after-image* until all undo entries are force-written to disk.
- No transaction commit unless all its associated undo and redo records are written to disk.

4.2.3 ARIES

The *Algorithms for Recovery and Isolation Exploiting Semantics* (ARIES) protocol is a more complex recovery protocol based on three main concepts:

- Just like in WAL, every operation must be first logged, then run on the database at commit time.
- All the operations prior to the crash are traced and transactions active at the time of the crash are undone, to bring the database back to the state it was in when the crash occurred.
- Changes performed during the undo phase are logged as well, so that crashes occurring during recovery do not lead to the repetition of undo operations.

Algorithm The actual ARIES recovery algorithm is run after system crashes and consists of three phases:

1. **Analysis Phase:** identifies which pages were being updated ("dirty" pages) and which transactions were active at the time of the crash.
2. **Redo Phase:** the uncommitted changes that are listed on the log but do not appear in memory are applied to the database.
3. **Undo Phase:** the changes brought by the transactions active at the time of the crash are read from the log in reverse order and undone.

Checkpoints Such an algorithm can be time-expensive. To reduce recovery time, checkpoints are introduced. Checkpoints are periodic "snapshots" of the state of a database and its active transactions which are included in the log.

A record is held in a separate, safe location, to store the LSN of the last checkpoint. This is the *Master Log Record* (MLR) and it significantly speeds up the analysis phase of the aforementioned algorithm. ARIES creates checkpoints in three steps:

1. A *begin-checkpoint* record is written to the log.
2. An *end-checkpoint* record is written to the log, along with the *active transactions* table and the *dirty pages* table.
3. After that, the MLR is updated with the LSN of the *begin-checkpoint* record.

5 Normalization

A *Normal Form* of a database is a restructuring of the database's schema that guarantees certain properties. Usually, normal forms are employed to reduce redundancy and solve update anomalies. The *normalization* of databases mainly revolves around the concept of functional dependency.

Definition (Functional Dependency). *Let r be a relation of schema $R(X)$ and let Y and Z be two non-empty subsets of X. A* functional dependency *(FD) exists between Y and Z if:*

$$\forall t_1, t_2 \in r.\ t_1[Y] = t_2[Y] \Rightarrow t_1[Z] = t_2[Z]$$

That is, any two tuples are the same on Z whenever they are the same on Y. We write $Y \to Z$.

For some trivial choices of attribute sets, this always happens. For example, $\forall Y \subseteq X, Z \subseteq Y.\ Y \to Z$. We are only interested in the non-trivial cases.

Definition (Non-Trivial FD). *A FD $Y \to Z$ is* non-trivial *if and only if no attribute $A \in Z$ appears in Y.*

Functional dependencies often entail anomalies, as they are a sign that heterogeneous, yet correlated data are part of the same relation (with the possibility of redundancy). This is not always the case, though. For example, for every K superkey of X and every subset Z of X we have that $K \to Z$, the intended behavior of a key.

5.1 Boyce-Codd Normal Form

The Boyce–Codd Normal Form (BCNF for short) is the strictest normal form that we see in this section.

Definition (BCNF). *A relation r is in* BCNF *if and only if for each non-trivial functional dependency $X \to Y$, X is a superkey of r.*

To BCNF-normalize a non-BCNF relation, we need to replace it with multiple relations, distributing FD across tables (where they will hopefully be allowed by the only BCNF constraint). This is not always possible, as loss of information may occur when splitting tables. We therefore enforce that our decompositions be *lossless*.

Definition (Lossless Decomposition). *A relation r can be* decomposed lossless *into two relations of attributes X and Y if and only if the joined projections of r on X and Y rebuild the original r. Algebraically, if and only if:*

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

This property is always verified when the intersection of X and Y constitutes a superkey for *at least* one of the decomposed tables. We also need our decompositions to be *dependency-preserving*.

Definition (Dependency-Preserving Decomposition). *A decomposition is said to be* dependency-preserving *if and only if the attributes of each FD of the original schema are found undivided in one of the decomposed tables.*

The process of normalizing a table to reach a BCNF involves identifying FDs and resolving them by splitting the table, provided that the lossless and dependency-preservation constraints are not violated. Note that this process is not always feasible (for some tables, a BCNF is not achievable).

5.2 Third Normal Form

The third normal form (3NF for short) is more relaxed than the BCNF. A consequence of such relaxation is that the 3NF has been proven to be always reachable from any table.

Definition (3NF). *A relation r is in* 3NF *if and only if for each non-trivial functional dependency $X \to Y$ at least one of the following is true:*

- *X is a superkey of r.*
- *Each attribute in Y is in at least one key of r.*

A 3NF-normalization involves decomposing a relation in a number of tables: one for each set of attributes which form a FD in the original schema. After that, every decomposed table is checked to see if it contains a key to the original relation.

5.3 Normalization in the E-R Model

Normalization can also happen at the level of the conceptual schema. In this context, the concept of implied functional dependencies becomes relevant.

Definition (Implication of Functional Dependencies). *Let F and D be two sets of functional dependencies. We say that F* implies *D if every relation that exhibits the FDs in F also exhibits those in D.*

Implied dependencies can be used as a criterion to split attributes over different entities or to reduce N-ary relationships. For example, if two members of a relationship depend on each other, one of them can be removed from the main relationship and put into a different relationship with the other.