Appendix 1

Data integrity

Mr Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Website ⇒ www.al-moualime.com)



University of Skikda — Department of Computer Science

1st Year Master NDS/AI

Advanced DataBases (ADB)

Updated on 2026-10-30 at 10:18:52



© 2026 Mr Laïdi FOUGHALI — www.al-moualime.com/adb

Creative Commons License — BY-NC-SA 4.0 International

Course conventions

Each page has a **colour** that shows its role:



Blue: the lecture, to read and to run.



Orange: an exercise or a recall; look for the answer.



Green: the solution of the exercise, on the next page.

- **Method**: answer the exercise **before** turning the page.
- **Review**: this link takes you to the lecture page that holds the answer; the **"↩ Back to slide N"** button (or the R key) brings you back to the exercise (in a PDF: Alt+←).
- **Interactive version**: the **Run** button runs the SQL example in the browser; "Reset database" restores the initial data.

Outline

1. Introduction
2. Expressing constraints in SQL
3. Going further
4. Analysis and checking
5. Triggers

Introduction

- A DBMS must ensure **data consistency** during updates.
- The data in a database are not independent, but obey semantic rules called **integrity constraints**.
- Integrity constraints can be declared explicitly or can be discreetly implicit.
- Constraints are validated at the end of each transaction by a non-violation check.

Validation techniques

1. **Prevention**: prevent invalid updates from happening.
2. **Detection**: undo incorrect transactions.
3. **Triggers**: operations fired by events associated with the database (*active database*).

Constraints: a sine qua non condition

What makes a DBMS *relational* *[Brouard, 2006]*

Support for **referential integrity** is the most important criterion for deciding whether a DBMS is truly relational. **Without this mechanism, a DBMS behaves like a mere file manager**: without constraints, it is no longer a relational database in the proper sense – just a pile of tables that look alike.

A rule of thumb for assessing a database

- **95 %** of tables should have a **primary key**.
- **80 %** should carry a **uniqueness constraint** (code, reference. . .).
- ≈**30 %** of columns should have a **validity constraint**.
- **All** relationships should have a **referential integrity** mechanism.

The figure that should make you think

Without constraints, the cost of cleaning data – observed in business intelligence projects (*data warehouse*) – can reach **up to 50 % of the overall cost** of a solution.

Examples of constraints

1. **Simple case**: equality of two data items — e.g. a foreign key `client_id` in `cars` must match an existing primary key `id` in `clients`.
2. **Complex case**: an assertion involving multiple data items.

Example (business rule)

The amount of unpaid orders must not exceed 20 % of the turnover already achieved by the customer.

Typology of constraints

- **Structural constraints**: "An integrity constraint specific to a model, expressing a fundamental property of a data structure of the model." The structures concerned are tables, columns and rows.
- **Non-structural constraints**: "An integrity constraint expressing an evolution rule that the data must satisfy during updates."

Notes

- Structural constraints are defined by `CREATE TABLE`.
- Non-structural constraints are defined by `CREATE ASSERTION` or `CREATE TRIGGER`.

Structural constraints

1. **Non-nullity** (`NOT NULL`): the value of an attribute must be provided.
2. **Key uniqueness** (`PRIMARY KEY`): defines non-null attributes as the primary key.
3. **Uniqueness** (`UNIQUE`): defines attributes as a candidate key.
4. **Referential** (`FOREIGN KEY`): the values of a foreign key must be equal to the values of a primary key or of a non-null candidate key.
5. **Domain** (`DOMAIN`): defines a user type.

Note

Some structural constraints can be described as behavioral constraints, for example key uniqueness.

Non-structural constraints (1/2)

For the relational model, these constraints can be expressed as first-order logic assertions. We distinguish:

1. **Functional dependencies**: $X \rightarrow Y$ (X determines Y) if for every value of X there is a single associated value of Y.
   Example: the student number determines a single last name.
2. **Multivalued dependencies** (generalization of the previous case): $X \twoheadrightarrow Y$ in a relation R if for every value of X there is a set of values of Y, independently of the other attributes.
3. **Inclusion dependencies**: the values of a group of columns of one table must remain included in those of a group of columns of another table — they generalize referential constraints.

Non-structural constraints (2/2)

4. **Temporal constraints**: involve time to track the updates of an attribute. *Example: a salary can only increase.*
5. **Equational constraints**: compare two arithmetic expressions computed from data, enforcing equality or inequality; the temporal dimension can come into play by comparing data before/after an update.

Note

Generalized dependencies group the various types of dependencies between attributes (functional, multivalued and inclusion), to better control redundancy.

Exercise 1: structural or non-structural?

Question: is each constraint **structural** or **non-structural**?

a) A customer's name must be filled in (`NOT NULL`).

b) A salary can only increase.

c) `customer_id` in `cars` must exist in `customers`.

d) Unpaid orders must not exceed 20 % of the customer's turnover.

Review: Typology of constraints

Solution of exercise 1

a) **Structural**: non-nullity, declared in `CREATE TABLE`.

b) **Non-structural**: a temporal constraint, a rule of evolution.

c) **Structural**: a referential constraint (`FOREIGN KEY`).

d) **Non-structural**: an assertion on multiple data (`CREATE ASSERTION` or a trigger).

Exercise 2: which dependency?

Question: name the type of each non-structural constraint: **functional**, **inclusion**, **temporal** or **equational**.

1. A student number determines a single surname.
2. The wine numbers of the `DRINKS` table must appear in the `WINES` table.
3. A salary can only increase.
4. The total of an invoice including tax equals the total before tax multiplied by 1.19.

Review: Non-structural constraints

Solution of exercise 2

1. **Functional dependency**: StudentNo $\rightarrow$ Name.
2. **Inclusion dependency**: it generalises the referential constraint.
3. **Temporal constraint**: it compares the value before and after the update.
4. **Equational constraint**: it enforces the equality of two arithmetic expressions.

Recall: the introduction

At the start of each section, three questions on the previous section.
Answer **from memory**, then click "Review" to check.

1. What are the three techniques for validating constraints?
   Review: Introduction

2. Which statements declare structural constraints? And non-structural ones?
   Review: Typology of constraints

3. What is a functional dependency?
   Review: Non-structural constraints

Expressing constraints in SQL (1/5)

```
CREATE TABLE <table name>
    ({<Attribute> <Domain> [<ATTRIBUTE CONSTRAINT>]}+)
    [<RELATION CONSTRAINT>]

<ATTRIBUTE CONSTRAINT> ::= NOT NULL | UNIQUE | PRIMARY KEY
    | REFERENCES <Relation> (<Attribute>) | CHECK <Condition>

<RELATION CONSTRAINT> ::= UNIQUE (<Attribute>+) |
    PRIMARY KEY (<Attribute>+) |
    FOREIGN KEY (<Attribute>+) REFERENCES <Relation> (<Attribute>+) |
    CHECK <Condition>
```

Expressing constraints in SQL (2/5)

```
CREATE TABLE DRINKS (
    ND INT NOT NULL,
    NW INT NOT NULL REFERENCES WINES(NW),
    DATE DEC(6) CHECK BETWEEN 010180 AND 311299,
    QUANTITY SMALLINT DEFAULT 1,

    PRIMARY KEY(ND, NW, DATE),
    FOREIGN KEY ND REFERENCES DRINKERS,
    CHECK (QUANTITY BETWEEN 1 AND 100)
);
```

Expressing constraints in SQL (3/5)

```
FOREIGN KEY (<Attribute>+)
    REFERENCES <Relation> (<Attribute>+)
    [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
    [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
```

Mode	**Explanation**
NO ACTION	No action — same effect as having no clause (SQL 1).
CASCADE	Propagates the action (DELETE/UPDATE) to the dependent tuples.
SET DEFAULT	Replaces the dependent foreign key with its default value.
SET NULL	Like SET DEFAULT, with the null value.

Expressing constraints in SQL (4/5)

```
CREATE ASSERTION <constraint name>
    [{BEFORE COMMIT |
      AFTER {INSERT|DELETE|UPDATE [OF (Attribute+)]} ON <Relation>}...]
    CHECK <Condition>
    [FOR [EACH ROW OF] <Relation>]
```

An assertion can be checked immediately after each specified update (`AFTER` clause), or at the end of the transaction (`BEFORE COMMIT` clause).

Expressing constraints in SQL (5/5)

Example 1: ensures that every wine has a strength greater than 10.

```
CREATE ASSERTION MINDEGREE BEFORE COMMIT
    CHECK (SELECT MIN(DEGREE) FROM WINES > 10)
    FOR WINES;
```

Example 2:

```
CREATE ASSERTION SUPQUALITY AFTER INSERT ON WINES
    CHECK ((SELECT COUNT(*)
    FROM DRINKS A, WINES V
    WHERE A.NW = V.NW
    AND V.QUALITY = "SUPERIOR") > 10).
```

Exercise 3: how many cars are left?

Question: how many cars are left after customer 1 is deleted?

```
DROP TABLE IF EXISTS cars, customers;
CREATE TABLE customers (id INT PRIMARY KEY, name TEXT);
CREATE TABLE cars      (plate       TEXT PRIMARY KEY,
                        customer_id INT REFERENCES customers(id) ON DELETE
     CASCADE);

INSERT INTO customers VALUES (1, 'Amina'), (2, 'Karim');
INSERT INTO cars      VALUES ('A-11', 1), ('A-12', 1), ('B-21', 2);

DELETE FROM customers WHERE id = 1;
SELECT COUNT(*) AS remaining FROM cars;
```

a) 3 **b)** 1 **c)** 0 **d)** an error

Choose, then run the script to check. Review: Referential constraint

Solution of exercise 3

Answer: b) 1 car.

- `ON DELETE CASCADE` **propagates** the deletion to the dependent rows.
- Deleting customer 1 also deletes their two cars, A-11 and A-12.
- Car B-21, Karim's, is left.

Exercise 4: and without an ON DELETE clause?

Question: same script, but the foreign key has **no** `ON DELETE` clause. What does the `DELETE` do?

```
DROP TABLE IF EXISTS cars, customers;
CREATE TABLE customers (id INT PRIMARY KEY, name TEXT);
CREATE TABLE cars      (plate       TEXT PRIMARY KEY,
                        customer_id INT REFERENCES customers(id));

INSERT INTO customers VALUES (1, 'Amina'), (2, 'Karim');
INSERT INTO cars      VALUES ('A-11', 1), ('A-12', 1), ('B-21', 2);

DELETE FROM customers WHERE id = 1;
```

a) it deletes customer 1 and their cars
b) it deletes customer 1 and keeps their cars
c) it is rejected: an error

Choose, then run the script to check. Review: Referential constraint

Solution of exercise 4

Answer: c) the `DELETE` is rejected.

- Without a clause, the mode is `NO ACTION`: nothing is done to the cars.
- Deleting customer 1 would leave A-11 and A-12 without a customer: the referential constraint would be violated.
- The DBMS therefore **rejects** the deletion; both customers stay in the table.

Recall: expressing constraints in SQL

Answer **from memory**, then click "Review" to check.

1. What is the difference between an **attribute** constraint and a **relation** constraint?
   Review: General syntax

2. What does `ON DELETE SET NULL` do?
   Review: Referential constraint

3. When is a `BEFORE COMMIT` assertion checked?
   Review: Independent constraint

SQL domains

Principle

[Brouard, 2006]

An **SQL domain** is a base type (INT, VARCHAR, ...) to which a default value, one or more CHECK constraints and an optional collation are added. It is then used like any other SQL type, in any table.

```
CREATE DOMAIN D_PERCENT AS FLOAT
    DEFAULT 0.0
    CHECK (VALUE BETWEEN 0.0 AND 100.0);

CREATE TABLE CUSTOMER (
    ID              INT NOT NULL PRIMARY KEY,
    NAME            VARCHAR(64),
    MAX_DISCOUNT    D_PERCENT NOT NULL DEFAULT 0.0
);
```

Benefit

All columns of the same domain (e.g. a percentage) automatically share the same constraint, without rewriting it table by table. **PostgreSQL** directly implements CREATE DOMAIN.

The CHECK constraint

Principle *[Brouard, 2006]*

`CHECK` restricts the acceptable values of one column (or several) by applying a **predicate**: a Boolean expression evaluated to TRUE, FALSE or NULL. If it is false for a row, that row is rejected.

The predicate may involve

explicit values, the `NULL` marker, functions or UDFs, arithmetic operators, comparisons, `AND`/`OR`/`NOT`, `BETWEEN`, `LIKE`, `IN`, `CASE`, and even subqueries.

Classic pitfall: on a **nullable** column, a `CHECK` must explicitly tolerate `NULL`:

```
CHECK (VALUE IS NULL OR VALUE BETWEEN 0 AND 100)
```

Otherwise, the column could never be left empty.

Multi-column foreign key: the MATCH clause

Purpose *[Brouard, 2006]*

The `MATCH` clause specifies how a foreign key spanning **several columns** behaves when some of them are `NULL`. By default (when not specified): `MATCH SIMPLE`. It is of no use for a single-column foreign key.

Mode	**Raises a violation if...**
`MATCH SIMPLE`	all columns are filled in **and** do not match.
`MATCH PARTIAL`	at least one filled-in value does not match.
`MATCH FULL`	the columns are neither all NULL, nor all filled in and correct.

Constraints: what varies across DBMSs

Constraint	Via	To check with the vendor
Domain	`CREATE DOMAIN`	Native support – PostgreSQL: yes.
Assertion	`CHECK`/triggers	**PostgreSQL does not implement** `CREATE ASSERTION` – use `BEFORE`/`AFTER` triggers instead (see Chapter 2).
Foreign key	`FOREIGN KEY`	`MATCH` and `ON UPDATE`/`ON DELETE` clauses.
`CHECK`	`CHECK`	Scope: a single row, or the whole table?

Key takeaway

This course uses **PostgreSQL**: the `CREATE ASSERTION` examples in this appendix are therefore **standard SQL2 that cannot be run as is** – in practice, they are rewritten as triggers (see the next section).

[Brouard, 2006]

Exercise 5: is the mark accepted?

Question: what does the second `INSERT` (mark 25) do?

```
DROP TABLE IF EXISTS exam_paper;
DROP DOMAIN IF EXISTS D_MARK;
CREATE DOMAIN D_MARK AS NUMERIC(4,2) CHECK (VALUE BETWEEN 0 AND 20);
CREATE TABLE exam_paper (id INT PRIMARY KEY, mark D_MARK);

INSERT INTO exam_paper VALUES (1, 15);
INSERT INTO exam_paper VALUES (2, 25);
SELECT COUNT(*) AS papers FROM exam_paper;
```

a) it inserts the mark 25
b) it inserts the mark reduced to 20
c) it is rejected: an error

Choose, then run the script to check. Review: SQL domains

Solution of exercise 5

Answer: c) the insertion is rejected; the table holds only one paper.

- The `mark` column belongs to the `D_MARK` domain: it inherits its `CHECK` constraint.
- 25 is not between 0 and 20: PostgreSQL reports that the value violates the domain's constraint.
- Any other column of this domain, in any table, would follow the same rule.

Exercise 6: MATCH SIMPLE or MATCH FULL?

Question: we insert a branch whose city is unknown (`NULL`). Which insertions are accepted?

```
DROP TABLE IF EXISTS branch_simple, branch_full, cities;
CREATE TABLE cities (country CHAR(2), city TEXT, PRIMARY KEY (country, city));
CREATE TABLE branch_simple (id INT PRIMARY KEY, country CHAR(2), city TEXT,
    FOREIGN KEY (country, city) REFERENCES cities MATCH SIMPLE);
CREATE TABLE branch_full   (id INT PRIMARY KEY, country CHAR(2), city TEXT,
    FOREIGN KEY (country, city) REFERENCES cities MATCH FULL);

INSERT INTO branch_simple VALUES (1, 'DZ', NULL);
INSERT INTO branch_full   VALUES (1, 'DZ', NULL);
```

a) both **b)** `branch_simple` only **c)** neither

Choose, then run the script to check. Review: The MATCH clause

Solution of exercise 6

Answer: b) only `branch_simple` accepts the row.

- `MATCH SIMPLE`: a violation only if **all** columns are filled in and differ; here `city` is `NULL`, so the row is accepted.
- `MATCH FULL`: the columns must be **all** `NULL` or **all** filled in and correct; (`'DZ'`, `NULL`) is rejected.
- PostgreSQL does not implement `MATCH PARTIAL`.

Recall: going further

Answer **from memory**, then click "Review" to check.

1. What is an SQL **domain**?
   Review: SQL domains

2. Does PostgreSQL implement `CREATE ASSERTION`? If not, what do you do?
   Review: What varies across DBMSs

3. What is the `MATCH` clause of a foreign key for?
   Review: The MATCH clause

Analysis of integrity constraints

Perform a consistency and non-redundancy test to answer the following questions:

1. Are the constraints consistent with each other?
2. Are they not redundant?
3. Which constraints must be checked after an insertion, a deletion, an update?
4. Is it possible to simplify the constraints to make checking easier?

Note

In practice, no DBMS (except perhaps the all-too-rare deductive DBMSs) is able to perform this test, let alone determine a minimal set of constraints.

Checking integrity constraints

In order to keep the database consistent, the DBMS must check that the integrity constraints are satisfied at the end of each transaction. In practice, this is done at each update.

- **Detection method**: find the tuples that do not satisfy a constraint after an update, and reject the update if any exist (**post-test**).
- **Prevention method**: prevent changes that would violate a constraint (**pre-test**).

On-the-fly checking of simple constraints (SQL 2)

On-the-fly prevention, just before the tuple is updated, by checking that the new value does not appear in an index, for:

1. Key uniqueness.
2. Referential integrity:
   - Insertion into the dependent table.
   - Update of the referencing column in the dependent table.
   - Deletion from the master table.
   - Key modification in the master table.

Note

Checks become complex in the case of cascading operations.

Exercise 7: detection or prevention?

Question: is each check a matter of **detection** (post-test) or **prevention** (pre-test)?

a) Before inserting, check in the index that the key does not already exist.

b) After an update, look for the rows that violate the constraint, and reject the update if there are any.

c) Refuse to delete a customer as long as cars reference it.

Review: Checking integrity constraints

Solution of exercise 7

a) **Prevention**: an on-the-fly check, just before the update.

b) **Detection**: the update happens, then it is checked and rejected if needed.

c) **Prevention**: the change that would violate the constraint is prevented.

Exercise 8: which operations to check?

Data: the foreign key `cars.customer_id` references `customers.id`.

Question: which operations must the DBMS check for this constraint?

a) an insertion into `cars`

b) an insertion into `customers`

c) a deletion from `customers`

d) a deletion from `cars`

e) a change of `cars.customer_id`

f) a change of `customers.id`

Review: On-the-fly checking

Solution of exercise 8

Answer: a), c), e) and f).

- **Dependent table** (`cars`): the insertion (a) and the change of the referencing column (e).
- **Master table** (`customers`): the deletion (c) and the change of the key (f).
- Inserting a customer (b) or deleting a car (d) cannot create an orphan reference.

Recall: analysis and checking

Answer **from memory**, then click "Review" to check.

1. What is the difference between the **detection** method and the **prevention** method?
   Review: Checking integrity constraints

2. Can a DBMS check that a set of constraints is consistent and non-redundant?
   Review: Analysis of integrity constraints

3. Which on-the-fly checks does referential integrity require?
   Review: On-the-fly checking

Triggers

- An **active** DBMS can fire an operation following a given event, forbid an operation, or send a message to the application.
- An **active database** moves part of the applications' semantics into the DBMS. Designing efficient active databases is a difficult problem.
- A **trigger** is a procedural rule fired by an event, in the form of an Event–Condition–Action triple:

```
WHEN <Event> IF <Condition on DB> THEN <Action on DB>
```

Note

Active DBMSs are therefore not standard.

Expressing triggers (SQL 3)

```
CREATE TRIGGER <name>
  -- event (with parameters)
  {BEFORE | AFTER | INSTEAD OF}          -- replaces a statement (security)
  {INSERT | DELETE | UPDATE [OF <column list>]}
  ON <table> [ORDER <value>]             -- priority
  [REFERENCING {NEW|OLD|NEW_TABLE|OLD_TABLE} AS <name>]...
  -- condition
  (WHEN (<SQL search condition>)
   -- action
   <SQL3 procedure>
   -- granularity
   [FOR EACH {ROW | STATEMENT}])
```

Triggers — integrity checking

```
-- Adding a drink
CREATE TRIGGER InsertDrink
BEFORE INSERT ON DRINKS REFERENCING NEW AS N
(WHEN (NOT EXIST (SELECT * FROM Wines WHERE NW=N.NW))
 ABORT TRANSACTION
 FOR EACH ROW) ;
-- Deleting a wine
CREATE TRIGGER DeleteWines
BEFORE DELETE ON WINES REFERENCING OLD AS O
(DELETE FROM DRINKS WHERE NW = O.NW
 FOR EACH ROW) ;
```

Triggers — temporal integrity checking

```
CREATE TRIGGER IncreasingSalary
BEFORE UPDATE OF Salary ON EMPLOYEE
REFERENCING OLD AS O, NEW AS N
(WHEN (O.Salary > N.Salary)
 SIGNAL.SQLState '7005' (''Salaries cannot decrease'')
 FOR EACH ROW);
```

Triggers — automatic key generation

```
CREATE TRIGGER SetWineKey
BEFORE INSERT ON PRODUCTS
REFERENCING NEW_TABLE AS N
(UPDATE N
SET N.NP = SELECT COUNT(*) +1 FROM PRODUCTS);
```

Triggers — managing aggregate data

```
CREATE TRIGGER TotalSal
AFTER UPDATE OF salary ON EMPLOYEE
REFERENCING OLD AS a, NEW AS n
(UPDATE TOTALS SET Increase = Increase + n.salary - a.salary
 WHERE ID = a.ID
 FOR EACH ROW);
```

Exercise 9: event, condition, action

Question: here is the "a salary cannot decrease" trigger, rewritten for PostgreSQL. Find the **event**, the **condition** and the **action**.

```
DROP TABLE IF EXISTS employee;
CREATE TABLE employee (id INT PRIMARY KEY, salary NUMERIC(8,2));

CREATE OR REPLACE FUNCTION f_increasing_salary() RETURNS trigger
LANGUAGE plpgsql AS $$
BEGIN
    IF NEW.salary < OLD.salary THEN
        RAISE EXCEPTION 'Salaries cannot decrease';
    END IF;
    RETURN NEW;
END;
$$;

CREATE TRIGGER increasing_salary
BEFORE UPDATE OF salary ON employee
FOR EACH ROW EXECUTE FUNCTION f_increasing_salary();
```

Review: Temporal integrity checking

Solution of exercise 9

Element	**In the trigger**
Event	`BEFORE UPDATE OF salary ON employee`, for each row
Condition	`NEW.salary < OLD.salary`: the new salary is smaller
Action	`RAISE EXCEPTION`: the update is rejected

In SQL3, the same rule is written `WHEN (O.Salary > N.Salary) SIGNAL ...`

Exercise 10: which final salary?

Data: the trigger of exercise 9 is in place.

Question: which salary does the last `SELECT` display?

```
DELETE FROM employee;
INSERT INTO employee VALUES (1, 3000);
UPDATE employee SET salary = 3500 WHERE id = 1;
UPDATE employee SET salary = 3200 WHERE id = 1;
SELECT salary FROM employee WHERE id = 1;
```

a) 3000 **b)** 3500 **c)** 3200

Choose, then run the script to check. Review: Temporal integrity checking

Solution of exercise 10

Answer: b) 3500.

- 3000 $\rightarrow$ 3500: the salary increases, the update is accepted.
- 3500 $\rightarrow$ 3200: the salary would decrease; the trigger raises an error and the update is cancelled.
- So the salary stays at 3500.

Any questions?