Course supplement

Data integrity

M. Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Website ⇒ www.al-moualime.com)

University of Skikda — Department of Computer Science

1st Year Master RSD/AI

Advanced Databases (BDDA)

Updated on 2026-09-22 at 22:26:47

Outline

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.

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).
```

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 supplement are therefore **standard SQL2 that cannot be run as is** – in practice, they are rewritten as triggers (see the next section).

[Brouard, 2006]

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.

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);
```

Any questions?