Integrity Constraints in SQL

Background

Chapter
About this Background Chapter

• In the main part of the TIS lecture, we will frequently discuss the problem of integrity of temporal relational databases. This additional set of slides has been designed in 2014.

• Thus, a good background in the syntax and semantics of integrity constraints in SQL is required.

• However, this feature of relational databases is (strangely?) neglected in most textbooks and lectures/courses on relational databases and SQL.

• Even in practice, just a fragment of the possibilities offered since long time by the SQL standard can be (and is) used. Most SQL DBMS don‘t support any kind of more complex (and thus more powerful) integrity constraints.

• Moreover, integrity checking (the automatic guarantee of satisfaction of all constraints in every state of a database) is not used by many application programmers even for the simple forms of integrity constraints supported in „real life“. This doesn‘t mean that there are no problems with integrity, however – the problem is just circumvented by many.
Quick Summary: Integrity in SQL

- There are two (more correctly: three) kinds of integrity constraints provided by the SQL standard:
  - **Table constraints:**
    - associated with a single table
    - declared within the CREATE TABLE statement for this table
    - checked on every modification of this table
    - potentially abbreviated as a column constraint if referring to a single column of the table
    - limited to few special cases using predefined keywords
  - ** Assertions:**
    - associated with several tables (or even views)
    - declared independently via a CREATE ASSERTION statement
    - checked on every modification of the entire database
    - using full expressivity of SQL’s query language, thus not limited to special cases
    - able to express every table constraint in „explicit“ form not using special syntax (keywords)

- Constraints are checked during transactions, i.e., sequences of modifications treated as a unit. Options of checking are IMMEDIATE or DELAYED (at transaction end).
Quick Summary: Table Constraints in SQL

- There are three main categories of table constraints in SQL:
  - NOT NULL constraints
  - CHECK constraints
  - KEY constraints

- **NOT NULL** constraints prevent any field in a particular column from being empty (in „SQL terminology“: from „containing“ a NULL value), e.g., NOT NULL (Name).

- **CHECK** constraints delimit the set of acceptable values for a particular column to those values from its value domain that satisfy a certain Boolean condition, e.g.,
  
  CHECK  Age BETWEEN 1 AND 130

(CHECK constraints containing nested subqueries are not accepted by most DBMS!)

- **PRIMARY KEY** and **UNIQUE** constraints ensure the uniqueness of values in (a combination of) columns of the table, thus preventing duplicate rows from existing, e.g.,
  
  PRIMARY KEY (MatrNr) or UNIQUE (Name, FirstName)

- **FOREIGN KEY** constraints ensure that all values in a particular (combination of) column(s) refers to the primary key values of another table, e.g.,
  
  FOREIGN KEY Participant REFERENCING Student
CREATE TABLE: Principle

- Most important DDL-operation: Creation of a new table

\[
\text{CREATE TABLE } \langle \text{table-name} \rangle \\
( \langle \text{lists-of-table-elements} \rangle ) \\
; \\
\]

- "Table elements" are
  - definitions of name and data type of each column, and
  - constraints referring to the newly created table.

- Syntax of a table definition:

\[
\text{CREATE TABLE } \langle \text{table-name} \rangle \\
\langle \text{column-name}_1 \rangle \langle \text{type}_1 \rangle \ [\langle \text{column-constraints}_1 \rangle ], \\
\langle \text{column-name}_2 \rangle \langle \text{type}_2 \rangle \ [\langle \text{column-constraints}_2 \rangle ], \\
\ldots \\
\langle \text{column-name}_n \rangle \langle \text{type}_n \rangle \ [\langle \text{column-constraints}_n \rangle ] \\
[\langle \text{table-constraints} \rangle ]
\]
CREATE TABLE: Example

Example:

SQL-statement defining a table `FootballMatch` containing the results of football matches in the national league:

```sql
CREATE TABLE FootballMatch
(
    Date date,
    HomeTeam text,
    GoalsH number(15) DEFAULT NULL CHECK ( >= 0 OR IS NULL),
    GuestTeam text,
    GoalsG number(15) DEFAULT NULL CHECK ( >= 0 OR IS NULL),
    Round number(15) NOT NULL CHECK ( > 0 AND < 35),
    PRIMARY KEY (Date, HomeTeam),
    FOREIGN KEY (HomeTeam) REFERENCES Teams,
    FOREIGN KEY (GuestTeam) REFERENCES Teams
) ;
```
CREATE TABLE: General Structure

Each table definition consists of **two parts**: The definitions of the individual columns, and (possibly) constraints valid for the entire table:

```sql
CREATE TABLE FootballMatch
(
  Date date,
  HomeTeam text,
  GoalsH number(15) DEFAULT NULL CHECK ( > = 0 OR IS NULL),
  GuestTeam text,
  GoalsG number(15) DEFAULT NULL CHECK ( > = 0 OR IS NULL),
  Round number(15) NOT NULL CHECK ( > 0 AND < 35),
  PRIMARY KEY (Date, HomeTeam),
  FOREIGN KEY (HomeTeam) REFERENCES Teams,
  FOREIGN KEY (GuestTeam) REFERENCES Teams
) ;
```
CREATE TABLE: Column Constraints

Each column definition itself consists of two parts, too:

- The declaration of a column name and a type of its values
- (Possibly) special constraints for the values in this column

Syntax of column constraints:

\[
\begin{align*}
\text{[ NOT NULL | UNIQUE ]} \\
\text{[ PRIMARY KEY ]} \\
\text{[ DEFAULT } \{ <\text{literal}> | \text{NULL} \} \text{ ]} \\
\text{[ REFERENCES } <\text{table-name}> \text{ ]} \\
\text{[ CHECK } <\text{condition}> \text{ ]}
\end{align*}
\]
CREATE TABLE: Table Constraints

The second part of a table definition is optional. It consists of one or more table constraints, normally expressing a restriction on several columns:

```
CREATE TABLE FootballMatch
(
    PRIMARY KEY (Date, HomeTeam),
    FOREIGN KEY (HomeTeam) REFERENCES Teams,
    FOREIGN KEY (GuestTeam) REFERENCES Teams
)
```

Syntax of table constraints:

```
[ UNIQUE ( <list-of-column-names> ) ]
[ PRIMARY KEY ( <list-of-column-names> ) ]
[ FOREIGN KEY ( <list-of-column-names> ) REFERENCES <table-name> ]
[ CHECK ( <condition> ) ]
```
Constraints in Table Definitions

- Table definitions (CREATE TABLE) contain two very similar kinds of constraints:
  - column constraints
  - table constraints (also called: row constraints)

- Column constraints are abbreviations of certain special forms of table constraints where the name of the resp. column remains implicit, e.g.

  - Column constraint:
    
    \[
    \text{Type} \quad \text{number}(15) \quad \text{CHECK (} > 0 \text{ AND } < 35 \text{ ),}
    \]

  - Table constraint:
    
    \[
    \text{CHECK (} \text{Type} > 0 \text{ AND } \text{Type} < 35 \text{ )}
    \]

- The condition part of such a CHECK constraint has to be satisfied in each admissible (legal, consistent) state of the database.
**UNIQUE and NOT NULL**

- **UNIQUE**-option: definition of a key (or: candidate key)
  - Single-column key:
    In a column definition: \(<\text{column-name}>\ldots\text{UNIQUE}\)
  - Multi-column key:
    Separate UNIQUE-clause as table constraint:
    \(\text{UNIQUE} \ (<\text{list-of-column-names}>\))

- **Semantics:** No two rows will ever have the same value in columns belonging to a key.

- **Exception:** Null values – NULL may occur several times in a UNIQUE-column.

- **Per table:** Arbitrarily many UNIQUE-declarations are possible.

- In a table with UNIQUE-declarations, no duplicates (identical rows) can exist!

- **Exclusion of null values** for individual columns: \(<\text{column-name}>\ldots\text{NOT NULL}\)
**PRIMARY KEY and DEFAULT**

- **Per table:** At most one (candidate) key can be declared the **primary key.**
  - single-column primary key:
    in column definition: `<column name> . . . PRIMARY KEY`
  - multi-column primary key:
    separate clause: `PRIMARY KEY ( <list-of-column-names> )`

- In addition: **No** column within a primary key may contain **NULL!**

- PRIMARY KEY **is not** the same as UNIQUE NOT NULL!
  (in addition: Uniqueness of the p. key within the table)

- Not a real „constraint", but rather similar in syntax:
  Declaration of a **default value** for columns of a table:
  Value which is automatically inserted if no explicit value is given during the insertion of a new row, e.g.:

```plaintext
Type number(15) DEFAULT 0
```
Foreign Key Constraints

• Second special form of constraint within a table declaration:

  ![Foreign key constraint](aka referential constraint)

• Situation: Column(s) of the table declared (called A) reference(s) (i.e., contains values of) a candidate key or primary key of another ("foreign") table B.

  Condition: A-columns contain only values actually occurring in the referenced B-column(s)!
Syntax of the corresponding constraint (as table constraint):

```
FOREIGN KEY ( <list-of-column-names> )
REFERENCES <table-name> [ ( <list-of-column-names> ) ]
```

If „target columns“ are missing:
primary key assumed

e.g.:

```
CREATE TABLE   t1
   (   a1 INT   PRIMARY KEY,
       ....
   )
```

`b1 references a1`

```
CREATE TABLE   t2
   (   b1 INT REFERENCES    t1,
       ....
   )
```
Referential Actions

• Complete syntax of a „referential constraint“ provides for various optional extensions:

\[
\text{FOREIGN KEY} \ (\ <\text{list-of-column-names}>\ )
\text{REFERENCES} \ <\text{base-table-name}> \ [\ (\ <\text{list-of-column-names}>\ )\ ]
\]

[ MATCH \ { FULL | PARTIAL } ]
[ ON DELETE \ { NO ACTION | CASCADE | SET DEFAULT | SET NULL} ]
[ ON UPDATE \ { NO ACTION | CASCADE | SET DEFAULT | SET NULL} ]

„Referential actions“ specify what happens in case of integrity violations

• Detailed discussion of all these extensions is beyond the scope of this short introduction.
Global Constraints in SQL

- Not supported by any commercial DB system till now, but defined in the SQL standard since long time:

- Assertions serve as a means for expressing **global integrity constraints** not tied to a particular table, but ranging over several tables.

- Syntax:

  ```sql
  CREATE ASSERTION <constraint-name>
  CHECK ( <conditional-expression> )
  ```

- In principle, assertions are sufficient for expressing all imaginable constraints, i.e., all "local" forms of constraints are redundant.

- On the other hand, many constraints can only be expressed via assertions, but not by means of table constraints.

- Example:

  ```sql
  CREATE ASSERTION lazy_professor
  CHECK EXISTS
  ( SELECT * FROM professor
    WHERE Name NOT IN ( SELECT Teacher
    FROM courses )
  );
  ```
Existential Conditions

- **Existential conditions** are used for checking whether the answer table of a subquery is empty, or not:

  ```sql
  SELECT Name
  FROM city
  WHERE EXISTS ( SELECT River
                  FROM city_at_river
                  WHERE City = Name )
  ```

  Which cities are situated close to a river?

- Existential conditions can be negated as well:  **NOT EXISTS**

- **Positive existential conditions** in queries are expressible without an explicit quantifier by eliminating the nesting of SELECT-blocks:

  ```sql
  SELECT Name
  FROM city, city_at_river
  WHERE City = Name
  ```

- This unnesting technique is **not** applicable for **NOT EXISTS** cases, however!

- And it doesn’t help for assertions, as they are „stand-alone“ conditions.
Simulation of FORALL via NOT EXISTS

• SQL has no keyword for universal quantification (no 'FORALL').

• Universal conditions have to be „simulated“ by means of logical transformations using double negation and existential quantification based on the following law of predicate logic:

\[ \forall x: F \iff \neg \exists x: \neg F \]

• Example: „Which river runs through every federal state in Germany?“

• In logic, this query can be formalized as follows:

\[ \{ x.River | \text{river-through-state}(x) \land \forall y: (\text{state}(y) \Rightarrow x.\text{State}=y.\text{Name} ) \} \]

• If no „forall“ is available, as in SQL, application of the above mentioned law results in the following more complex formulation:

\[ \{ x.River | \text{river-through-state}(x) \land \neg \exists y: \neg (\text{state}(y) \Rightarrow x.\text{State}=y.\text{Name} ) \} \]
• Applying two more transformation laws of propositional logic eliminates the implication and pushes the inner negation even more inward, thus resulting in a slightly more intuitive formalization:

\{x.\text{River} \mid \text{river\_through\_state}(x) \land \lnot \exists y: (\text{state}(y) \land \lnot\text{x.\text{State}}=\text{y.\text{Name}})\}\n
• If this simple query is to be expressed in SQL, it is necessary to go exactly this way (involving quite a bit of logic) in order to be able to „simulate“ FORALL:

```
(SELECT X.\text{River}
FROM river\_through\_state AS X
WHERE NOT EXISTS
  ( SELECT *
    FROM state AS Y
    WHERE NOT X.\text{State}=Y.\text{Name} )
)
```
• Each SQL column and table constraint (both of which are in fact implemented by all relational DBMS, even though not completely), is a shorthand for an assertion, e.g.

```sql
CREATE TABLE students
    (MatrNr integer NOT NULL CHECK > 0, ...
)
```

which itself is short for

```sql
CREATE TABLE students
    (MatrNr integer, ..., CONSTRAINT correct_matr_nr
        CHECK (MatrNr IS NOT NULL AND MatrNr > 0))
```

• Using an assertion, we have to make explicit the very idea of a table constraint, namely to forbid the occurrence of any rows violating the resp. constraint:

```sql
CREATE ASSERTION correct_matr_nr AS CHECK NOT EXISTS
    (SELECT * FROM students
        WHERE NOT (MatrNr IS NOT NULL AND MatrNr > 0))
```
Assertions in SQL (2)

• For expressing a (candidate or primary) key constraint (UNIQUE or PRIMARY KEY, resp., in SQL) as an assertion, a much more complex explicit condition is required – that’s why SQL offers the shorthand for this frequently used feature:

```sql
CREATE TABLE students
(MatrNr integer UNIQUE,
Name text, Sem integer)
```

• If a column A has key property in a table T, its values uniquely identify exactly one of the rows of T in every state of the DB. That is: If two rows of T have the same A-value, the rows must be equal in all other columns, too, i.e., they must be identical:

```sql
CREATE ASSERTION unique_matr_nr AS
CHECK NOT EXISTS
(SELECT * FROM students AS S1
WHERE EXISTS (SELECT * FROM students AS S2
WHERE S1.MatrNr = S2.MatrNr
AND NOT (S1.Name = S2.Name AND
S1.Sem = S2.Sem)))
```
As long as assertions are not supported by commercial DBMS, DB users/administrators have to take care of more complex aspects of the integrity of their data manually:

**Data curation**

A convenient way to do this is to use views searching for rows violating an intended, but not implemented assertion (and reestablishing integrity manually afterwards):

- In case of CHECK **NOT EXISTS** condition: Search for rows satisfying condition, in order to take care of getting rid of all rows satisfying condition.
- In case of CHECK **EXISTS** condition: Search for rows satisfying condition, too, in order to re-establish integrity by adding at least one row satisfying condition.

```sql
CREATE VIEW check_unique_matr_nr AS
    (SELECT * FROM students AS S1
     WHERE EXISTS (SELECT * FROM students AS S2
                    WHERE S1.MatrNr = S2.MatrNr
                    AND NOT (S1.Name = S2.Name AND
                              S1.Sem = S2.Sem)))
```

Better: select all combinations of S1/S2 such that ….
Integrity Checking in SQL

• Important topic related to SQL constraints:
  Modalities of checking for constraint violations

• Changes in SQL are usually part of greater units of change called transactions:
  • Transaction: Sequence of DML statements viewed as „indivisible units"
  • Transactions are either executed completely, or not at all!
  • Transactions always have to lead to consistent DB states satisfying all integrity constraints stated in the resp. DB schema.
  • More detailed discussion of the concept „transaction“: later!

• Important motivation for introducing transactions:
  Some transitions from a consistent state into a consistent follow-up state are only possible via inconsistent intermediate steps!

• Consequence for integrity checking during transaction processing:
  Checking of constraints should (more or less always) take place at the end of a transaction!
Integrity Checking in SQL (2)

• **In SQL however:** Unless defined otherwise, integrity checking always happens *immediately* (i.e., directly after the execution of each update).

• **Motivation:** Many simple table constraints can and ought to be checked immediately as they are independent of any other updates.

• But in particular for „referential cycles“:

  Checking at transaction end is **inevitable!**

  - **C1:** „Each course is given by a professor!"
  - **C2:** „Each professor has to give at least one course!"

  ![Course-Professor Diagram]

  When hiring a new professor a consistent state can be reached *only* via a transaction consisting of two individual insertions:

  INSERT INTO professor
  INSERT INTO course

  Each intermediate state would be inconsistent: **No sequence possible!**
Thus: two forms of integrity checking in SQL

**IMMEDIATE** and **DEFERRED**

Meaning: IMMEDIATE-constraints are *immediately* checked, for DEFERRED-constraints checking is deferred to the end of the current transaction.

Unfortunately: Without explicitly stating one of these alternatives, IMMEDIATE is assumed (which somehow contradicts the idea of a transaction).

This default assumption can be changed for individual constraints by declaring them as

**INITIALLY DEFERRED.**

„INITIALLY“, because the checking status can be changed dynamically during a running transaction:

```
SET CONSTRAINTS { <list-of-constraints> | ALL }
{ DEFERRED | IMMEDIATE }
```

In addition: Some constraints can be declared **NOT DEFERRABLE.** But the even more important **NOT IMMEDIATE** does not exists in SQL!

In summary: Integrity checking in „full" SQL can be a difficult affair!
Update Operations in SQL: Overview

- SQL offers three basic operations for changing data:
  - **INSERT**  insertion of rows
  - **UPDATE**  modification of values in columns
  - **DELETE**  deletion of rows

- All three types of update operation can be combined with queries for retrieving the rows of a particular table to be inserted/updated/deleted.

- Reminder: There is the danger of a **terminology conflict**:
  - „Update“: in colloquial DB-style refers to any kind of change
  - **UPDATE** in SQL: means column value replacement only
**INSERT**

- Format of insertions:

\[
\text{INSERT INTO } <\text{table-name}> \ [ ( <\text{list-of-columns}> ) ] \ <\text{table-expression}>
\]

- Two variants:
  - **Direct** reference to one or more rows to be inserted, e.g.
    \[
    \text{INSERT INTO professors (Name, Rank, Department) VALUES ("Manthey", "C3", "III")}
    \]
  - **Indirect** identification of the rows to be inserted via a query, e.g.
    \[
    \text{INSERT INTO professors}
    \text{SELECT * FROM researchers AS R WHERE R.qualification = "PhD"}
    \]
UPDATE and DELETE

- Format of modifications:
  
  ```
  UPDATE <table-name>
  SET <list-of-assignments>
  [ WHERE <conditional-expression> ]
  ```

- Modifies all rows of "table name" satisfying the WHERE-part according to the assignments of values to columns given in the SET-part.

- Syntax of an individual assignment:

  ```
  <column-name> = { <scalar-expression> | DEFAULT | NULL }
  ```

- Example:

  ```
  UPDATE professors
  SET Name = 'N.N.'
  WHERE Dept = 'II'
  ```

- Quite similar: Deletions

  ```
  DELETE FROM <table-name>
  [ WHERE <conditional-expression> ]
  ```