SQL: Constraints

- *Foreign Keys*
- *Local and Global Constraints*
- *Triggers*

And a precursor to *transactions*.

Version

initial	*2018-03-11*
adapted	*2020-11-08*
last revised	*2020-11-10*

Printable version of talk

- Follow this link:
  SQL: Constraints [to **pdf**].
- Then *print* to get a **PDF**.

(*This only works correclty in **Chrome** or **Chromium**.*)

Acknowledgments

Thanks to

- *Jeffrey D. Ullman*
  for initial slidedeck
- *Jarek Szlichta*
  for the slidedeck with significant refinements
- *Wenxiao Fu*
  for conversion to **Reveal.js** and significant refinements

Constraints → Used to specify rules for data in table
Triggers → executed when a specified condition occurs.

Constraints and Triggers

- A **constraint** is a relationship among data elements that the DBMS is required to enforce.
  - Example: *key* constraints
- **Triggers** are only executed when a specified condition occurs, e.g., insertion of a tuple.
  - Easier to implement than complex constraints.

Types of Constraints in SQL

- **keys** (*entity integrity*)
- **foreign keys** (*referential-integrity*)
- **value-based** constraints.
  - Constrain values of a particular attribute.
- **tuple-based** constraints.
  - Relationships among components.
- **Assertion** constraints: (Not same as java)
  any SQL Boolean expression.

Keys

Which we all know and love!

SQL allows us to define an attribute or attributes to be *a* key for a relation with the constraint keywords `primary key` or `unique`.

Single-Attribute Keys

Place `primary key` or `unique` after the type in the declaration of the attribute.

```
CREATE TABLE Beers (
     name   CHAR(20) PRIMARY KEY,
     manf   CHAR(20)
);
```

```
CREATE TABLE Beers (
     name   CHAR(20) UNIQUE,
     manf   CHAR(20)
);
```

Are these the same?

→ Unique is an alternative to primary key with a caveat.

Technically, a *primary-key* constraint is the combination of a *not-null* constraint and a *unique* constraint.

Single-Attribute Keys (single primary key)

```
CREATE TABLE Student (
    name   CHAR(20),
    st#    INTEGER    PRIMARY KEY,
    sin    INTEGER    UNIQUE,
           ⋮
);
```

Multi-attribute Keys (multiple primary key)

The *bar* and *beer* together make the key for *Sells*:

```
CREATE TABLE Sells (
    bar    CHAR(20),
    beer   VARCHAR(20),
    price REAL,
    PRIMARY KEY (bar, beer)
);
```

Foreign Keys

déjà vu!

Values appearing in attributes of one relation must appear together in certain attributes of another relation.

- An attribute, or set of attributes, is a foreign key if it references some attribute(s) of a second relation.
- This represents a constraint between relations.

E.g., in $\mathbf{Sells}(\mathrm{bar}, \mathrm{beer}, \mathrm{price})$, we might expect that a beer value also appears in $\mathbf{Beers}.\mathrm{name}$.

Expressing Foreign Keys

Foreign key → used to link 2 tables

Use keyword `references` either

1. after an attribute (for one-attribute keys) or
2. As an element of the schema:

```
FOREIGN KEY (<list of attributes>)
  REFERENCES <relation> (<attributes>)
```

the table it is being linked to

Referenced attributes must be declared `primary key` or `unique`. *Why?*

Values of a foreign key must also appear in the referenced attributes of some tuple.

Example: with attribute

```
CREATE TABLE Beers (
    name CHAR(20) PRIMARY KEY,
    manf CHAR(20)
);
CREATE TABLE Sells (
    bar   CHAR(20),
    beer  CHAR(20)
        REFERENCES Beers(name),
    price REAL
);
```

link between 2 tables

numeric data type

the complete example

Example: As Schema Element

```
CREATE TABLE Beers (
    name CHAR(20) PRIMARY KEY,
    manf CHAR(20)
);
CREATE TABLE Sells (
    bar   CHAR(20),
    beer  CHAR(20),
    price REAL,
    FOREIGN KEY(beer)
        REFERENCES Beers(name)
);
```

Enforcing Foreign-Key Constraints

If there is a foreign-key constraint from relation **R** referencing relation **S**, two types of violations are possible:

1. An `insert` or `update` to **R** introduces new vaues that are not found in **S**. (insert/update in one table introduces new values to linked table)
2. A `delete` or `update` to **S** causes some tuples of **R** to "dangle." (delete/update causes referencing nothing in the linked table)

dangle → referencing nothing.

Action Taken (1)

Example: Suppose $\mathbf{R} = \mathbf{Sells}, \mathbf{S} = \mathbf{Beers}$.

- An insert or update to $\mathbf{Sells}$ that introduces a nonexistent beer must be rejected.
- A deletion or update to $\mathbf{Beers}$ that removes a beer value found in some tuples of $\mathbf{Sells}$ can be handled in one of *three* ways.

Action Taken (2) A

How to accommodate for the changes made in **Beers** in **Sells**?

- **Default**: *reject* the modification.
- **Cascade**: "cascade" the same changes to **Sells**.
  (Same changes to the linked table)
  - *deleted* beer: delete matching **Sells** tuples.
  - *updated* beer: change value in **Sells**.
- **Set NULL**: Change the (problematic) beer value in **Sells** to NULL.
  - In this case with **Sells**, would this strategy work?

Upto here on 12/11/20

Example for Cascade

Delete the *Bud* tuple from **Beers**:

- Then delete all tuples from **Sells** that have `beer='Bud'`.

Update the *Bud* tuple by changing `'Bud'` to `'Budweiser'`:

- Then change all **Sells** tuples with `beer='Bud'` to `beer='Budweiser'`.

Example for Set NULL

Delete the *Bud* tuple from **Beers**:

- change all tuples of **Sells** that have `beer='Bud'` to have `beer=NULL`.

Update the *Bud* tuple by changing `'Bud'` to `'Budweiser'`:

- same changes as for deletion.

Choosing a Policy

When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates.

Follow the foreign-key declaration by:

```
ON [UPDATE | DELETE][SET NULL | CASCADE]
```

Two such clauses may be used. Otherwise, the default (reject) is used.

Example for Setting Policy

```
CREATE TABLE Sells(
    bar   CHAR(20),
    beer  CHAR(20),
    price REAL,
     FOREIGN KEY (beer)
          REFERENCES Beers(name)
          ON DELETE SET NULL
          ON UPDATE CASCADE
);
```

Check → Used to limit the value range that can be placed in a column

Attribute-Based Checks

Constraints on the value of a particular attribute.

- Add `CHECK(<condition>)` to the declaration for the attribute.
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery.

Example: Attribute-Based Check

```
CREATE TABLE Sells(
     bar    CHAR(20),
     beer   CHAR(20)
         CHECK ( beer IN
             ( SELECT name
               FROM Beers ) ),
     price REAL CHECK (price <= 5.00 )
);
```

Data type

Timing of Checks

Attribute-based checks are performed only when a value for that attribute is inserted or updated.

- `CHECK (price <= 5.00)` checks every new price and rejects the modification (for that tuple) if the price is more than $5.
- `CHECK (beer IN (SELECT name FROM Beers))` not checked if a beer is deleted from Beers (unlike foreign-keys)!

Tuple-Based Checks

`CHECK (<condition>)` may be added as a relation-schema element.

- The condition may refer to any attribute of the relation. But other attributes or relations require a subquery
- Checked on insert or update only.

Example: Tuple-Based Check

Only Joe's Bar can sell beer for more than $5:

```
CREATE TABLE Sells (
      bar    CHAR(20),
      beer   CHAR(20),
      price REAL,
      CHECK (bar = 'Joe''s Bar'
          OR price <= 5.00)
 );
```

Assertion → Piece of sql which makes sure a condition is satisfied or it stops taking action on a database object.

Assertions

These are database-schema elements, like relations.

Defined by:

```
CREATE ASSERTION <name> CHECK (<condition>
```

Condition may refer to any relation or attribute in the database schema.

Example: Assertion

In **Sells**(bar, beer, price), no bar may charge an average of more than $5.

```
CREATE ASSERTION NoRipoffBars CHECK(
   NOT EXISTS (
         SELECT bar
         FROM Sells
         GROUP BY bar
         HAVING 5.00 < avg(price)
   )
);
```

Example: Assertion

In **Drinkers**(name, addr, phone) and **Bars**(name, addr, license), there cannot be more bars than drinkers.

```
CREATE ASSERTION FewBar CHECK(
    (SELECT COUNT(*) FROM Bars) <=
    (SELECT COUNT(*) FROM Drinkers)
);
```

Timing of Assertion Checks

In principle, we must check every assertion after every modification to any relation of the database.

A clever system can observe that only certain changes could cause a given assertion to be violated.

- E.g., No change to **Beers** can affect **FewBar**. Neither can an insertion to **Drinkers**.

Different Constraint Types

Type	When Declared	When Activated	Guaranteed to hold?
Attribute CHECK	with attribute	on insertion or update	not if subquery
Tuple CHECK	relation schema	insertion or update to relation	not if subquery
Assertion	database schema	on change to any relation mentioned	yes

Modification of Constraints

Giving Names to Constraints

Add the keyword `CONSTRAINT` and then a name:

```
name  CHAR(20) CONSTRAINT beer_pk PRIMARY KEY,
```

```
price REAL CONSTRAINT price_ck CHECK (price <= 5.00 ),
```

```
CONSTRAINT bar_price_ck CHECK (bar = 'Joe''s Bar'
                                OR price <= 5.00)
```

It's a good habit to give each of your constraints a name even if you do not believe you will ever need to refer it.

Alter → changes the table value type.

Altering Constraints

```
ALTER TABLE Sells DROP CONSTRAINT price_ck
ALTER TABLE Sells DROP CONSTRAINT bar_pric

ALTER TABLE Sells ADD CONSTRAINT price_ck
   CHECK (price <= 5.00);
ALTER TABLE Sells ADD CONSTRAINT bar_price
   CHECK (bar = 'Joe''s Bar' OR price <= 5.
```

The added constraint must be of a kind that can be associated with tuples, such as tuple-based constraints, key, or foreign-key constraints. `price_ck` and `bar_price_ck` are all tuple-based constraints now. We cannot bring them back as attribute-based constraints.

Triggers: Motivation

- Assertions are powerful, but the DBMS often cannot tell when they need to be checked.
- Attribute- and tuple-based checks are checked at known times, but are not powerful.

Triggers let the user decide when to check for any condition.

Event-Condition-Action Rules

Another name for “trigger” is ECA rule, or *event-condition-action* rule.

- **Event**: typically a type of database modification,
  - e.g., "`insert on Sells`"
- **Condition**: any SQL boolean-valued expression.
- **Action**: any SQL statements.

Preliminary Example: A Trigger

Instead of using a *foreign-key* constraint and rejecting insertions into **Sells**(bar, beer, price), with unknown beers, a trigger can add that beer to **Beers**, with a null manuf.

```
1. CREATE TRIGGER BeerTrig
2. AFTER INSERT ON Sells
3. REFERENCING NEW ROW AS NewTuple
4. FOR EACH ROW
5. WHEN (NewTuple.beer NOT IN
            (SELECT name FROM Beers))
6. INSERT INTO Beers(name) VALUES (NewTuple.beer);
```

2 ⇒ The event; 5 ⇒ The condition; 6 ⇒ The action.

Note: Triggers can reject insertion/update.

The Syntax for Trigger

```
1. Create Trigger <name>
2. {BEFORE | AFTER} {<trigger event>}
                           ON <Table name>
3. [REFERENCING <old or new values alias l
4. [FOR [EACH] {ROW | STATEMENT}]
5. [WHEN (condition)]
6. <triggered SQL statement>
```

Options: The Trigger Event

```
<trigger event> ::=
INSERT |
DELETE |
UPDATE [ OF <trigger Column list> ]
```

`<trigger Column list>` could be one or more columns.

Options: FOR [EACH] ROW | STATEMENT

Triggers are either "row-level" or "statement-level".

- `FOR EACH ROW` indicates row-level
  - *row-level* triggers : execute once for each modified tuple.
- `FOR EACH STATEMENT` indicates statement-level (default).
  - *statement-level* triggers : execute once for a SQL statement, regardless of how many tuples are modified.

Options: REFERENCING

```
[REFERENCING <old or new values alias list>]

<old or new values alias> ::=
OLD [ ROW ] [ AS ] old values <Correlation name> |
NEW [ ROW ] [ AS ] new values <Correlation name> |
OLD TABLE [ AS ] <old values Table alias>        |
NEW TABLE [ AS ] <new values Table alias>
```

- `INSERT` statements imply a new tuple (for row-level) or table (for statement-level).
  - The "table" is the set of inserted tuples.
- DELETE implies an old row or table.
- UPDATE implies both.

Options: The Condition

Any boolean-valued condition.

- Evaluated on the database as it would exist before or after the triggering event, depending on whether BEFORE or AFTER is used.
  - but always before the changes take effect.
- Access the new/old row/table through the names in the REFERENCING clause.

Options: The Action

```
<triggered SQL statement> ::=
        SQL statement |
        BEGIN {SQL statement;}... END
```

- There can be more than one SQL statement in the action.
    - state them in the BEGIN...END one by one
- But queries make no sense in an action, so we are really limited to modifications.

Another Example

Using **Sells**(bar, beer, price) and a unary relation **RipoffBars**(bar), maintain a list of bars that raise the price of any beer by more than $1.

```
1. CREATE TRIGGER PriceTrig
2. AFTER UPDATE OF price ON Sells
3. REFERENCING
        OLD ROW as ooo
        NEW ROW AS nnn
4. FOR EACH ROW
5. WHEN (nnn.price > ooo.price + 1.00)
6. INSERT INTO RipoffBars VALUES (nnn.bar);
```

2 ⇒ The event - only changes to prices; 3 ⇒ Updates let us talk about old and new tuples;

4 ⇒ We need to consider each price change; 5 ⇒ Condition: a raise in price > $1;

6 ⇒ When the price change is great enough, add the bar to RipoffBars

The Syntax for Trigger in PostgrSQL

```
1. Create Trigger <name>
2. {BEFORE | AFTER} {<trigger event>}
                         ON <Table name>
3. [REFERENCING <old or new values alias list> ]
4. [FOR [EACH] {ROW | STATEMENT}]
5. [WHEN (condition)]
6. EXECUTE {FUNCTION | PROCEDURE} <function_name (arguments)>
```

Different from the standard SQL syntax, CREATE TRIGER in PostgreSQL requires put the action into a function or procedure.