- candidate key: unique
  - example:
    FirstName character(20) not null,
    LastName character(20) not null,
    unique (LastName, FirstName)

Referential Integrity

- foreign key: foreign key
- unique foreign key models a 1:1-relationship
- Foreign key must be marked as primary key in the referenced relation.
- example:

```sql
create table customers(cid integer, ..., primary key(cid))
create table orders(..., cid integer, ..., foreign key(cid) references customers(cid))
```
- standard behavior when violating a referential integrity constraint: rejection of the action that triggered the violation
- If a foreign key is specified with a cascade clause, changes of the respective primary key are propagated.
example

+ `create table lectures (...

    foreign key held_by integer references professors(pers-id)
    on delete cascade
    on update cascade);

+ The deletion of a tuple with the key `pers-id` in `professors` has an effect on `lectures`. Also there the tuple is then deleted. This enables the realization of a dependent relationship.

+ A change of the attribute `pers-id` in `professors` is propagated to the foreign key `held_by` in `lectures`.

example

+ `create table lectures (...,

    foreign key held_by integer references professors(pers-id)
    on delete set null
    on update set null);

+ The attribute value `held_by` in `lectures` is set to `null`, if the referenced tuple with the key `pers-id` in `professors` is deleted.

+ The attribute value `held_by` in `lectures` is set to `null`, if the referenced tuple with the key `pers-id` in `professors` is changed.
7.4 Management of Integrity Constraints

Adding ICs

- `alter table orders`  
  `add constraint limit check(price * number < 10000)`

- `alter table customers`  
  `add constraint unique_name unique cname`

- `alter table Kunde`  
  `drop constraint unique_name`

Enabling and disabling ICs

- `alter table customers enable constraint unique_name`
- `alter table customers disable constraint unique_name`
7.5 Complex Integrity Constraints

Assertions

- An **assertion** is a predicate that expresses a condition which is to be always satisfied by a database. Domain constraints and referential ICs are special kinds of assertions. There are also conditions which cannot be expressed with these two kinds like conditions with respect to several relations.

- syntax: `create assertion <assertion name> check <condition>`

- example: There must be at least four professors in order to maintain teaching.
  
  ```sql
  create assertion AlwaysFourProfessors
  check (4 <= (select count(*) from professors))
  ```

- The DBMS tests an assertion for validity. If the assertion is valid, future modifications of the database are only allowed if the assertion is not violated.

- Complex assertions can lead to an overhead.

- Assertions should be used with care.
**Trigger**

- **A trigger**
  - is a user-defined procedure which is automatically executed from the DBMS if a certain condition is fulfilled or as a side effect of a modification of the database.
  - answers to events with respect to a given relation.
  - is a general and powerful mechanism for maintaining data consistency.
  - can take not only check functions but also computation functions.
  - can update statistics, for example, or compute the values of derived columns.
  - consists of a head and a PL/SQL block.
  - contains in its head preconditions for executing the block.
  - is not part of SQL92. We orient ourselves to Oracle.

- **Two requirements for the design of a trigger mechanism**
  - specification of conditions when the trigger is to be executed
  - specification of the actions that have to be performed if a trigger is executed

- **example: trigger preventing that professors can be demoted by a rank**

  ```sql
  create trigger noDemotion
  before update on professors
  ```
for each row
\[\text{when} \ (\text{:old.rank is not null})\]
\[\text{begin}\]
\[\text{if} \ \text{:old.rank} = \text{"C3"} \text{ and } \text{:new.rank} = \text{"C2"} \text{ then } \text{:new.rank} = \text{"C3"} \text{ end if;}\]
\[\text{if} \ \text{:old.rank} = \text{"C4"} \text{ then } \text{:new.rank} = \text{"C4"} \text{ end if;}\]
\[\text{if} \ \text{:new.rank is null then } \text{:new.rank} = \text{:old.rank} \text{ end if;}\]
\[\text{end}\]

Construction of a trigger head

- Creation and change, resp., of an existing trigger with the DDL command

\text{create trigger} \ <\text{name}> \ \text{resp.} \ \text{replace trigger} \ <\text{name}>\]

- time of releasing the trigger body before or after the operation which released the trigger

\text{[before} \ | \ \text{after]}\]

- trigger event

\text{update} \ [\text{of} \ <\text{column}_1, \ \text{column}_2, \ ...>] \ \text{on} \ <\text{relation name}>
**insert on** <relation name>

**delete on** <relation name>

A trigger can be defined for one or several events. In case of several events, a case distinction can be expressed in the body through the clauses:

if updating [column₁, column₂, ...] then ...

if inserting then ...

if deleting then ...

- **trigger type**
  - [for each row]
    - **Command-oriented** trigger (default) are released exactly once either before (before) or after (after) the respective event.
    - **Row-oriented** trigger are called for each changed tuple. In the body one has the possibility (and only this one) to address the old resp. the new value of the tuples of the relation over :old or :new for update, over :new für insert and over :old for delete.

Another access to the relation is not possible any more, even if the relation would be addressed in the respective block.
trigger restriction

- **when** <predicate>
- Conditions can be formulated which release the execution of the trigger body.
- If a row-oriented trigger is used, the new resp. old tuple of the relation can be addressed by the keywords *new* resp. *old*.

Trigger body

- procedure definition as PL/SQL-Block with the aforementioned extensions

Examples

- Protocol of the changes of the attribute *salary* of a relation *Persons*

```sql
create trigger StoreSalary
before update on Persons
for each row when (:old.salary > 1500)
begin insert into diff values (:old.salary, :new.salary, sysdate) end;
```
check at insertion time that a salary increase is inapplicable to persons with a high salary

```sql
create trigger CheckSalary
before update on Persons
for each row when (:new.salary > 1500)
begin
    :new.salary := :old.salary; // Zuweisung nur möglich bei before update
end;
```

Problems when applying triggers

- User must control that triggers do not contradict each other.
- A trigger can activate another trigger. Cycles should be avoided.
- termination of events
- If a consistency condition can be formulated by an integrity constraint, triggers should not be used.
7.6 Integrity constraints in Query-By-Example

Model inherent integrity concepts

- QBE supports key integrity and domain constraints
- check of key integrity when inserting a data record
- change of key attributes impossible

Explicit integrity constraints

- For each relation a “constraint table” exists in which the ICs can be inserted as rows.

```
<table>
<thead>
<tr>
<th>R</th>
<th>Attr_1</th>
<th>Attr_2</th>
<th>...</th>
<th>Attr_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.CONSTR(&lt;trigger&gt;).I.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

<trigger> is a list of elements out of {I., U., D.} as well as possibly user-defined triggers. The columns can contain entries of the form \( \theta c \) (\( \theta \) comparison operator, \( c \) constant), also example elements for links with other tuples, and simple constants.
example: No balance may fall under USD -100.

<table>
<thead>
<tr>
<th>customer</th>
<th>cname</th>
<th>caddr</th>
<th>account</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.CONSTR(I., U.).I.</td>
<td></td>
<td></td>
<td>≥ -100</td>
</tr>
</tbody>
</table>

The account of Jones may not be overchecked.

<table>
<thead>
<tr>
<th>customer</th>
<th>cname</th>
<th>caddr</th>
<th>account</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.CONSTR(I., U.).I.</td>
<td>Jones</td>
<td></td>
<td>≥ 0</td>
</tr>
</tbody>
</table>

For each order the customer name must be contained in the relation *customer* and the product must be contained in the relation *supplier*.

<table>
<thead>
<tr>
<th>order</th>
<th>cname</th>
<th>product</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.CONSTR(I.).I.</td>
<td>_N</td>
<td>_W</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>customer</th>
<th>cname</th>
<th>caddr</th>
<th>account</th>
</tr>
</thead>
<tbody>
<tr>
<td>_N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>supplier</th>
<th>sname</th>
<th>saddr</th>
<th>product</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>_W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analogously, one would have to insert a constraint into the *customer* relation that a tuple may not be deleted, if an order of this customer still exists.

**User-defined triggers**

- example:

```
<table>
<thead>
<tr>
<th>customer</th>
<th>cname</th>
<th>caddr</th>
<th>account</th>
</tr>
</thead>
<tbody>
<tr>
<td>JonesIC</td>
<td>Jones</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I.CONSTR(JonesIC).l.</td>
<td></td>
<td></td>
<td>≥ 0</td>
</tr>
</tbody>
</table>
```

The first row defines a trigger “JonesIC”, which becomes active each time the tuple for Jones is changed. The integrity constraint of the second row is only checked in this case.

**Dynamic integrity constraint**

- example: The bread price may not be increased.

```
<table>
<thead>
<tr>
<th>supplier</th>
<th>sname</th>
<th>saddr</th>
<th>product</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.CONSTR(U.).I.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_N</td>
<td>bread</td>
<td></td>
<td>≤ _P</td>
<td></td>
</tr>
<tr>
<td>_N</td>
<td>bread</td>
<td></td>
<td>_P</td>
<td></td>
</tr>
</tbody>
</table>
```

new value

old value
8. Design Theory for Relational Databases

8.1 Introduction

- so far: methodical design from an E-R schema to a relational database schema

- Questions
  - What does a good conceptual database schema look like?
  - How can the quality of a database schema be assessed?

- hence: conceptual fine tuning of the generated relational database schema on the basis of formal methods
  - functional dependencies as generalization of the key concept
  - multi-valued dependencies as generalization of functional dependencies
  - normal forms for relation schemas on the basis of functional dependencies
    + goal: estimation of the “quality” of a relation schema
    + If a relation schema does not fulfil these normal forms, so-called normalization algorithms can be applied to it in order to decompose it into several new schemas where each resulting schema fulfils the corresponding normal form.
example:

- database schema 1

  \textit{customer(cname, caddr, account)}
  
  \textit{order(cname, product, amount)}
  
  \textit{supplier(sname, saddr, product, price)}

- database schema 2

  \textit{customer-addr(cname, caddr)}
  
  \textit{customer-account(cname, account)}
  
  \textit{order(cname, product, amount)}
  
  \textit{supplier(sname, saddr)}
  
  \textit{offer(sname, product, price)}

problem discussion of database schema 1 by an example

- \textit{supplier(sname, saddr, product, price)}
  
  + \textbf{redundancies}
    
    For each product the address of the supplier is stored.
  
  + \textbf{update anomalies}
    
    The address of a supplier can be changed in one of its tuples, but remain unchanged in another tuple (→ inconsistency).
Design Theory for Relational Databases

8. Design Theory for Relational Databases

8.1 Introduction

+ **insertion anomalies**
  A supplier’s address cannot be inserted without a product.

+ **deletion anomalies**
  The deletion of the last product offered by a supplier leads to a loss of the supplier’s address.
  - intuitive improvement
    
    \[
    \text{supplier}(\text{sname}, \text{saddr})
    \]
    
    \[
    \text{offer}(\text{sname}, \text{product}, \text{price})
    \]
  - advantages
    + no redundancy
    + no anomalies
  - But: For finding the supplier’s address of a product, an expensive join is needed.

**Design goals**

- avoidance of redundancy and anomalies
- avoidance of the problems of information representation
- avoidance of information loss
- maybe consideration of efficiency aspects