9.4 Management of Integrity Constraints

Adding ICs

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

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

- `alter table customers`  
  `drop constraint unique_name`

Enabling and disabling ICs

- `alter table customers enable constraint unique_name`
- `alter table customers disable constraint unique_name`
9.5 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
when (:old.rank is not null)
begin
    if :old.rank = "C3" and :new.rank = "C2" then :new.rank = "C3" end if;
    if :old.rank = "C4" then :new.rank = "C4" end if;
    if :new.rank is null then :new.rank = :old.rank end if;
end
```

Construction of a trigger head

- Creation and change, resp., of an existing trigger with the DDL command
  
  ```sql
  create trigger <name> resp. replace trigger <name>
  
  time of releasing the trigger body before or after the operation which released the trigger
  [before | after]
  ```
trigger event

update [of <column₁, column₂, ...>] on <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 for 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;    // assignment only possible for 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.
9.6 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>Attr1</th>
<th>Attr2</th>
<th>...</th>
<th>Attrn</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.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(l., 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(l., 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(l.).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></td>
<td></td>
<td></td>
<td>_W</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).I.</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>_N</td>
<td>bread</td>
<td>≤ _P</td>
<td></td>
</tr>
<tr>
<td>_N</td>
<td>bread</td>
<td>_P</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
13. Object-Relational Databases

13.1 Why object-relational databases?

- Shortcomings of the relational model, legacy data models (e.g., hierarchical model, network model) and query languages like SQL92 especially for new, emerging applications with complex objects, e.g., computer-aided desktop publishing, image representation for satellite images or weather forecasting, biological and genomic information, spatial, spatiotemporal and geographical data in maps, data about air and water pollution, traffic data, audio and video data, etc.

- Legacy systems have concepts to explicitly model relationships. But they make extreme use of pointers in their implementation (navigation) and do not have concepts like object identity, inheritance, encapsulation or support for object types and complex objects.

- Object-relational database systems (ORDBs) as a way to enhance the capabilities of the relational model by object-oriented features.

- ORDBs as a compromise between relational database systems with their shortcomings for complex objects and object-oriented database systems with their lacking acceptance.
13.2 Object-relational support in SQL3

- SQL3 = SQL2 + object-oriented and other features

Objects in SQL3

- SQL allows **user-defined types (UDTs)**, **type constructors**, **collection types**, **user-defined functions**, support for **large objects**, **triggers**

- Objects in SQL3 can be of two kinds:
  - **Row types** or **tuple types** whose instances are tuples in relations
  - **Abstract data types (ADTs)** that are types whose internal structure is hidden and that can be used as components of tuples

- A row type is defined as
  ```sql
  create type <row_type_name> [as [row]] (<component declarations>);
  ```

- Examples
  ```sql
  create type address_type as (street varchar(35), city varchar(25), zipcode char(5));
  create type employee_type as (name varchar(35), addr address_type, age int);
  ```