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
  ```sql
  [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 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

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>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></td>
<td></td>
<td></td>
<td></td>
<td>_W</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></td>
<td>_N</td>
<td>bread</td>
<td>_P</td>
<td></td>
</tr>
</tbody>
</table>

  new value

  old value
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);
  ```
Use of the common dot notation to refer to components: addr.city

- An **array type** may be specified as in
  
  ```sql
  create type company_type as
      (comp_name varchar(20), location varchar(20) array[10]);
  ```
  
  location[1] refers to the first value in the array

- Corresponding relations can be created by
  
  ```sql
  create table employee of employee_type;
  create table company of company_type;
  ```

- Use of an explicit **row constructor**
  
  ```sql
  create table employee as
      (name varchar(35),
       addr row(street varchar(35), city varchar(25), zipcode char(5)),
       age int);
  ```

- Insertion
  
  ```sql
  insert into employee values
      (‘John Smith’, row(‘Mall Avenue’, ‘Gainesville’, ‘32711’), 36);
  ```
**insert into** company **values**
   (‘XYZ’, **array**[‘Mall Avenue’, ‘Sales Street’, ‘Sellers Drive’]);

Alternatively:

**insert into** company **values** (‘XYZ’, **array**[]);

**update** company
   **set** location = **array**[‘Mall Avenue’, ‘Sales Street’, ‘Sellers Drive’]
   **where** comp_name = ‘XYZ’;

Later:

**update** company
   **set** location[2] = ‘Salesman Street’ **where** comp_name = ‘XYZ’;

- An extension allows one to specify that and how **object identifiers** are created.

**create table** employee **of** employee_type **ref is** emp_id **system generated**;
**create table** company **of** company_type
   (**ref is** comp_id **system generated**, **primary key** (comp_name));

Syntax: **ref is** <oid_attribute> <value_generation_method>

Attribute named **oid_attribute** is used to identify individual tuples in the table. DBMS generates a unique identifier for each tuple.
A component attribute of one tuple may be a reference (specified by using the keyword ref) to a tuple of another table.

Example

```plaintext
create type employment_type as
  (empl ref (employee_type) scope(employee) references are checked,
   comp ref (company_type) scope(company) references are checked);
create table employment of employment_type;
```

Keyword scope specifies the name of the table whose tuples can be referenced by the reference attribute. This is similar to a foreign key. But the system generated value is used rather than the primary key.

Keyword references are checked indicates that dangling references (i.e., invalid reference values) are not allowed.

Path expressions are built by applying the dot notation. However, for an attribute whose type is ref, the dereferencing symbol -> is used (similar as in the C programming language).
Example

```sql
select e.empl->name
from employment as e
where e.comp->comp_name = 'XYZ';
```

*a* a reference to a tuple *t*, *a* attribute in *t*, *r->a* denotes value of *a* in *t*.

- Object identifiers can be explicitly declared in the type definition rather than in the table declaration.

- Example (change of the definition of *employee_type*)

```sql
create type employee_type as
    (name char(35), age int, emp_id ref (employee_type));
create table employee of employee_type
values for emp_id are system generated;
```

- **Encapsulation** of operations in SQL

  User can create a named *user-defined type* with own behavioral specification by specifying *user-defined functions* (methods, operations) in addition to the attributes.

- In SQL3, a UDT (ADT) is generally defined through
  - the specification of a set of declarations for stored attributes that represent the value of the UDT,
Operations defining equality (equal) and an order (less than) on the UDT,
- Operations defining the behavior of the ADT.

Example: Specify a method for “Extract the apartment number from a string that forms the street attribute of the address_type row type declared before.”

```java
create type address_type as (street varchar(35), city varchar(25), zipcode char(5));
method apt_no() returns char(8);
```

The code for implementing the method still has to be written:

```java
method
create function apt_no() returns char(8) for address_type as
external name '/x/y/aptno.class' language 'java';
```

Java is the implementation language, the code is stored in a file with the specified pathname.

UDT can have a number of user-defined functions associated with it. Syntax:

```java
method <name>(<argument_list>) returns <type>;
```
Built-in functions for UDTs

- **Constructor function** `type_t()` creates and returns a new object (instance) of type `type_t`
- **Observer function** `A` is implicitly created for each attribute `A` to read its value. `A(X)` or `X.A` returns the value of attribute `A` of `type_t` if `X` is of type `type_t`.
- **Mutator function** sets the value of the attribute to a new value (update).

Two types of functions: internal SQL and external

- **Internal functions** are written in the extended SQL/PSM (Persistent Stored Modules) language (not discussed in this class, similar to Oracle’s PL/SQL).
- **External functions** are written in a host language (e.g., Java, C++). Only their signature (interface) appears in the UDT definition. Syntax:
  ```
  declare external <function_name> <signature> language <language_name>;
  ```

Many ORDBMs provide the user with packages of **abstract data types** (ADTs). They are purchased separately from the basic system.

- Data Blades in Informix Universal Server
- Data Cartridges in Oracle
- Extenders in DB2
UDTs can be used as attribute types in SQL and as parameter types in a function or procedure.

Encapsulation of components (attributes and functions) at different levels

- **public**: These components form the interface of the ADT and are visible outside the ADT definition for all authorised users.
- **private**: These components are totally encapsulated and are only visible within the definition of the ADT containing them.
- **protected**: These components are partially encapsulated. They are visible within their ADT and within the definition of all subtypes of the ADT.

**Example**

```sql
CREATE TYPE employee_type
(  
  public
    name char(29), b_address address_type, manager employee_type,
    hiredate date;
  private
    base_salary decimal(7, 2), commission decimal(7, 2);
  public
    function working_years (p employee_type) returns int
  < program code to compute the number of working years >
```
public
  function working_years (p employee_type, y years) returns employee_type
  < program code to update the number of working years >
public
  function salary (p employee_type) returns decimal
  < program code to compute the salary of an employee >
);

Data types for large unstructured complex objects

- Data type blob for binary large objects
- Data type clob for character large objects
- Example
  create table employees (resume clob(75K), signature blob(1M), picture blob(12M));
- only read and write operations for byte ranges available, no random access to sub-structures
Inheritance

- regarding types and relations expressed by means of the `under` keyword
- Example regarding relations
  ```
  create table person (name char(20), sex char(1), age integer);
  create table employee under person as (salary float);
  create table customer under person as (account integer);
  
  Employee inherits all attributes (and methods) of person and has an additional attribute salary.
  ```
- Rules
  - All attributes are inherited.
  - The order of supertypes in the `under` clause determines the inheritance hierarchy.
  - An instance of a subtype can be used in every context in which a supertype instance is used.
  - A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
  - When a function is called, the best match is selected based on the types of all arguments.
13.3 Informix Universal Server

Introduction

- Combination of relational and object database technologies of two previously existing products
  - relational DBS *Informix*
  - Illustra, originated from the *Postgres* DBMS as a research project at the University of California at Berkeley

- Extension of the relational data model by
  - support for additional or *extensible data types*
  - support for user-defined routines (procedures or functions)
  - inheritance
  - support for indexing extensions
  - *Data Blades* API
Extensible data types

- Informix Universal Server = basic DBMS + Data Blade Modules
- Number of pre-defined data types
  - types for two-dimensional geometric objects (points, lines, circles, ellipses), images, time series, text, Web pages
  - when the Informix Universal Server was announced, 29 Data Blades existed
- Application may create own types, thus making the data type notion fully extendible.
- four constructs to declare additional data types
  - opaque type
  - distinct type
  - row type
  - collection type
- Opaque type
  - Internal representation is hidden and used for encapsulating a type.
  - User has to provide casting functions to convert an opaque object between its hidden representation in the server data base and the visible representations on the client.
− User functions *send/receive* to convert the internal server representation to/from the client representation

− Specification of an opaque type includes name, fixed internal length, maximal variable length, alignment (which is the byte boundary), hashing possible or not

  ```sql
  create opaque type fixed_opaque_udt (internallength = 8, alignment = 4, cannothash);
  create opaque type var_opaque_udt (internallength = variable, maxlen = 1024, alignment = 8);
  ```

  − Distinct type
    − An existing type is extended through inheritance. The newly defined type inherits the functions/methods of its base type, if they are not overwritten.

  ```sql
  create distinct type hiring_date as date;
  ```

  − Row type
    − represents a composite attribute, analogous to a *struct* type in C
    − support of inheritance by using the keyword *under*

  ```sql
  create row type employee_type(ename varchar(25), address varchar(200), salary int);
  create row type engineer_type(licence varchar(20)) under employee_type;
  ```
create row type engr_mgr_type(manager_start_date varchar(10), dept Managed varchar(20)) under engineer_type;

- application, e.g. create table employee of type employee_type;

Collection type
- Collections include lists (list), sets (set) and multisets (multiset) of built-in types and user-defined types.
- Example
  
  create table employee(name varchar(50) not null, commission multiset(money));

Support of user-defined routines

- user-defined functions to manipulate user-defined types
- Implementation as stored procedures or in C or in Java
- Example:

  create function equal(arg1 fixed_opaque_udt, arg2 fixed_opaque_udt)
  returning boolean;
  external name “/usr/lib/informix/libopaque.so (fixed_opaque_udt_equal)”
  language C;
  end function;
Support for inheritance

- two kinds of inheritance
  - data (attribute) inheritance
  - function (operation) inheritance

- Data inheritance
  - creation of sub types with the **under** keyword
  - enable the specification of type hierarchies
  - subtype inherits all attributes of the superior types up to the root

- Example

  ```sql
  create row type employee_type (ename varchar(25); ssn char(9); salary int);
  create row type engineer_type (degree varchar(10); licence varchar(20)) under employee_type;
  create row type engr_mgr_type(start_date varchar(10); dept_managed varchar(20)) under engineer_type;
  ```

Type **engineer_type** is subtype of type **employee_type**, inherits all attributes of **employee_type**, and has two additional attributes.
Function inheritance

- Functions can also be inherited.
- Example:

```sql
create function overpaid(employee_type)
returns boolean as
return $1.salary > (select salary from employee where ename = "Bill Brown")
```

- All tables under the `employee` table automatically inherit this function.
- Redefinition for `engr_mgr_type`:

```sql
create function overpaid(engr_mgr_type)
returns boolean as
return $1.salary > (select salary from employee where ename = "Jack Jones")
```

- The query

```sql
select g.ename from engineer as g where overpaid(g);
```

uses the first definition of `overpaid` while the query

```sql
select gm.ename from engr_mgr as gm where overpaid(gm);
```

uses the second definition of `overpaid`, which overrides the first definition.

- This is called **operation (function) overloading** (special kind of polymorphism).
Support for index extensions

- Example:
  The query
  \texttt{create index empl\_city on employee (city(address));}
  creates an index on the table \textit{employee} using the value of the \textit{city} function.

- In order to support user-defined index structures, operator classes are available that support user-defined data types in a generic B-tree.

- R-trees are available for geometric data types.

Support for Data Blades API

- purpose: definition of new data types and functions for new kinds of applications

- Two-dimensional spatial data types for geometric applications, e.g.
  - A \textit{point} is defined by \((X, Y)\) coordinates.
  - A \textit{line} is defined by its two end points.
  - A \textit{polygon} is defined by an ordered list of \(n\) points that form its vertices.
  - A \textit{path} is defined by a sequence (ordered list) of points.
  - A \textit{circle} is defined by its center point and its radius.
Functions on these data types comprise e.g.

- distance: \( \text{point} \times \text{point} \rightarrow \text{float} \)
  
  distance: \( \text{point} \times \text{line} \rightarrow \text{float} \)
  
  distance: \( \text{point} \times \text{polygon} \rightarrow \text{float} \)
  
  distance: \( \text{line} \times \text{line} \rightarrow \text{float} \)
  
  ...

- overlap: \( \text{polygon} \times \text{polygon} \rightarrow \text{bool} \)
  
  overlap: \( \text{line} \times \text{polygon} \rightarrow \text{bool} \)
  
  ...

- contains: \( \text{point} \times \text{polygon} \rightarrow \text{bool} \)
  
  contains: \( \text{point} \times \text{circle} \rightarrow \text{bool} \)
  
  ...

Image data types

- Data type \text{image} offered with a large variety of standard storage formats, support of the formats tiff, gif, jpeg, photoCD, group 4, fax

- Operations e.g.

  rotate: \( \text{image} \times \text{angle} \rightarrow \text{image} \)
  
  crop: \( \text{image} \times \text{polygon} \rightarrow \text{image} \)
  
  flip: \( \text{image} \rightarrow \text{image} \)
  
  plus: \( \text{image} \times \text{image} \rightarrow \text{image} \)
minus: \( image \times image \rightarrow image \)
intersection: \( image \times image \rightarrow image \)
union: \( image \times image \rightarrow image \)

- further data types e.g. for time series, text