8. More SQL features:
Views, PL/SQL, Functions,Triggers

8.1 Views and view updates
8.2 Application architectures
8.3 PL/SQL – PL/pgSQL
8.4 Functions and Procedures
8.5 Triggers
8.6 Abstract Data Types

see Kemper/Eickler chap. 14, Elmasri chap. 6.8, O'Neill: Chap. 4,
Melton: SQL99, Postgres and Oracle Manuals (PL/PGSQL,PL/SQL)

8.1 Views

Def.: A **view** is a **named SQL**-query, which becomes part of the schema as a virtual table

Intention

- Casting the database schema for different applications
- Access protection
- Privacy
- Structuring of SQL programs

⇒ The RDM concept for external schemas ("3-schema-architecture")

Materialized Views

Def.: A **materialized view** is a **temporary Table** , which contains the result set of an SQL query

- Not in all DBMS
- Often used in **replication** scenarios
- No way to insert / delete data
- But refreshing of the view makes sense
- Sometimes called **snapshot**

- Different from **temporary tables**

```
CREATE TEMPORARY TABLE Temp AS (<Query>)
```

- Insertion / Deletion allowed
- Dropped at the end of a session

SQL Views

May be defined on **base tables** (ordinary tables)

or on **views** (or both)

```
CREATE VIEW LargeCities
 (name,population,country, code,fraction)
AS
(SELECT ci.name, ci.population, co.name, co.code,
  ci.population/co.population
 FROM City ci JOIN Country co ON ci.country = co.code
 WHERE ci.population > 1000000)
```

```
CREATE VIEW VeryLargeCities   AS
 (SELECT name, population, country
  FROM LargeCities l
  WHERE l.population >= 3000000)
```

implicite column names

Views and privacy

Very large American cities:
JOIN with `encompasses(continent,country...)`

```
CREATE OR REPLACE VIEW VLAmeriCities AS
(SELECT c.name, c.population, c.country
  FROM LargeCities c JOIN Encompasses e
  ON c.code =e.country
  WHERE e.continent = 'America'
  AND c.population >= 3000000)
```

Views may be used **like ordinary table in queries.**

Privacy: column access may be granted even if access to base table is not allowed !

Views and code readability

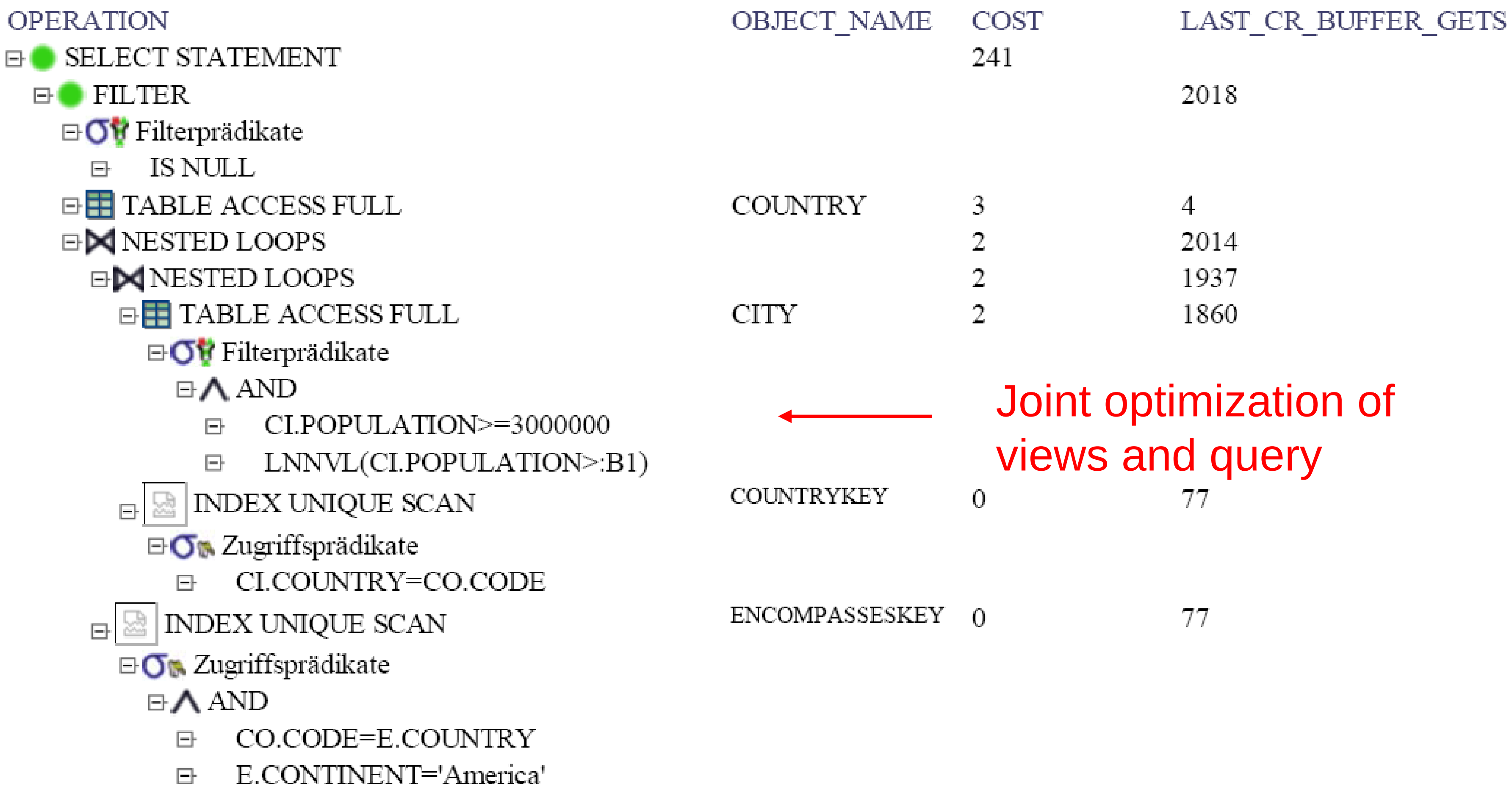
.. simplify SQL queries

Countries having more inhabitants than all american big cities

```
SELECT c.name,  c.population
FROM  country c
WHERE c.population < ALL(SELECT  population
                          FROM VLAmeriCities)
```

Operator tree of query more complicated...

Query plan



```
OPERATION                                        OBJECT_NAME     COST   LAST_CR_BUFFER_GETS
⊟ ● SELECT STATEMENT                                             241
  ⊟ ● FILTER                                                            2018
    ⊟ Filterprädikate
      ⊟   IS NULL
    ⊟ TABLE ACCESS FULL                          COUNTRY         3      4
    ⊟ NESTED LOOPS                                               2      2014
      ⊟ NESTED LOOPS                                             2      1937
        ⊟ TABLE ACCESS FULL                      CITY            2      1860
          ⊟ Filterprädikate
            ⊟ AND
              ⊟   CI.POPULATION>=3000000
              ⊟   LNNVL(CI.POPULATION>:B1)
        ⊟ INDEX UNIQUE SCAN                      COUNTRYKEY      0      77
          ⊟ Zugriffsprädikate
            ⊟   CI.COUNTRY=CO.CODE
      ⊟ INDEX UNIQUE SCAN                        ENCOMPASSESKEY  0      77
        ⊟ Zugriffsprädikate
          ⊟ AND
            ⊟   CO.CODE=E.COUNTRY
            ⊟   E.CONTINENT='America'
```

Joint optimization of views and query

Evaluation of views

Steps:

[1. Transform query on view using its definition]

2. Construct operator tree including view definitions and query
3. Optimize plan
4. Execute query on base tables

Views in Postgres

More general substitution concept in Postgres

Rules are "first class objects": **`CREATE RULE...`**

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

equivalent to

```
CREATE TABLE myview (<same column list as mytab>);

CREATE RULE "_RETURN" AS ON SELECT TO myview DO
  INSTEAD SELECT * FROM mytab;
```

Kind of dynamic view evaluation compared to static rewrite of query or query tree

8.2 Updatable views

View updates

Many views are **not updatable**. Obviously:

```
CREATE OR REPLACE VIEW PopulInCities (country,
cityPop)
AS
(SELECT co.name, sum(ci.population)
 FROM City ci JOIN Country co ON
ci.country=co.code
 GROUP BY co.name)
```

View not updatable if defined using:

- Aggregation
- Arithmetic in Projection
- DISTINCT

Semantic characterization of updatable views

Def: A **view V is updatable** if for every update u (*) there exist one or more updates c_u which applied to the base relations and the subsequent application of the view definition result in the same result:

$$u\,(V(D)) \;=\; V\,(c_u\,(D)\,)$$

- Semantic characterization,
- Wanted: **syntactic criteria** for updatability

(*) as if it were materialized

Syntactic criteria

Read only views may be arbitrarily defined,
Update is rejected, if view not updatable.

Syntactic criteria

Not updatable (SQL 92)

- if grouped (GROUP BY), HAVING or aggregated
- DISTINCT in SELECT clause
- set operators (INTERSECT, EXCEPT, UNION)
- more than one table in FROM clause
- **No updates on join views** (restrictive!)

Views and joins

```
CREATE VIEW CCP AS
(SELECT c.name, c.capital, ci.population
 FROM Country c JOIN City ci
  ON c.capital=ci.name and c.code=ci.country
WHERE ci.population > 1000000
ORDER BY c.name)
```

Base tables: Country, City,

Join on key: **row insertion** in one table (`Country`) may generate one new row in in the other (`City`), if not already present.

Syntactic criteria (2)

SQL 1999

Columns (of views) are **potentially updatable** if ...

- no DISTINCT operator
- no GROUP BY, HAVING clause
- no derived columns (e.g. arithmetic expressions)

(1) Column is updatable if potentially updatable and one table in FROM clause (!)

Key preserved tables

… SQL 1999: more than one table in FROM clause

(2) Column c is **updatable** if potentially updatable and

- c belongs to exactly one table

- the **key** of the table is **preserved**, i.e. the update of c may be traced back to exactly one row.

Table is **key preserved** if every key of the table can also be a key of the join result table.
A key-preserved table has its keys preserved through a join.

Find updatable columns

Find updatable columns by querying the catalogue

```
SELECT column_name, updatable
FROM user_updatable_columns
WHERE table_name ='LARGECITIES'
-- Oracle
```

This is a (system) view

must be upper case

```
COLUMN_NAME                    UPDATABLE
------------------------------ ---------
NAME                           YES
POPULATION                     YES
COUNTRY                        NO
CODE                           NO
FRACTION                       NO
```

Views WITH CHECK OPTION

Issue: **side effects** on base table rows, no effect on view

```
CREATE VIEW CCLarge(ctryName, capital,  population) AS
  (SELECT c.name as ctryName, c.capital, ci.population
   FROM Country c JOIN City ci
ON c.capital=ci.name and c.code=ci.country
   and c.province = ci.province
  WHERE ci.population > 1000000)
  WITH CHECK OPTION
```

```
UPDATE TABLE CC_Large
SET  population = population - 20000
WHERE capital = 'Amsterdam'  --has 1011000 inhabitants
```

What happens?

CHECK OPTION

Update may result in insertion and deletion (!) of rows

CHECK OPTION: update and insert must result in rows the **view can select** , otherwise exception raised

Example above: update has to be performed on base table

View update by triggers

Triggers: Event – Condition – Action rules

Event: **`Update, insert, delete`** (basically)

Condition: **`WHEN`** < some conditon on table>

Action: some operation (expressed as DML, DB-Script language expression, even Java)

INSTEAD OF Triggers (Postgres: rules)

- defined on views
- specify what to do in case of an update of the view

details on triggers: see below

Summary views

- Views: important mechanism for
  access protection / privacy
  simplyfy SQL application programming

- **The** mechanism for defining external schemas in the RDM
- Useful for modeling **generalization hierarchies**
- Disadvantage: **updates** (inserts, deletes) not always possible
- Criteria for updatable views complex
- **INSTEAD OF triggers** are a convenient work around

8.2 Application Architectures

- SQL is an **interactive language**, but...
- Main usage: access database from application program

  Means basically: SQL-statements statically known, but parameterized:

  ```
  SELECT name INTO  :ctryName
  FROM Country JOIN Economy ON...
  WHERE gdp < :threshold
  ```

  "Impedance mismatch": tuple sets vs records or objects
- Typical database usage:
  independent applications concurrently access DB
- Web based user interface is standard today

  ⇒ **Big differences of (application) system architectures**

Business logic

Big question: where sits the **"business logic" ?**

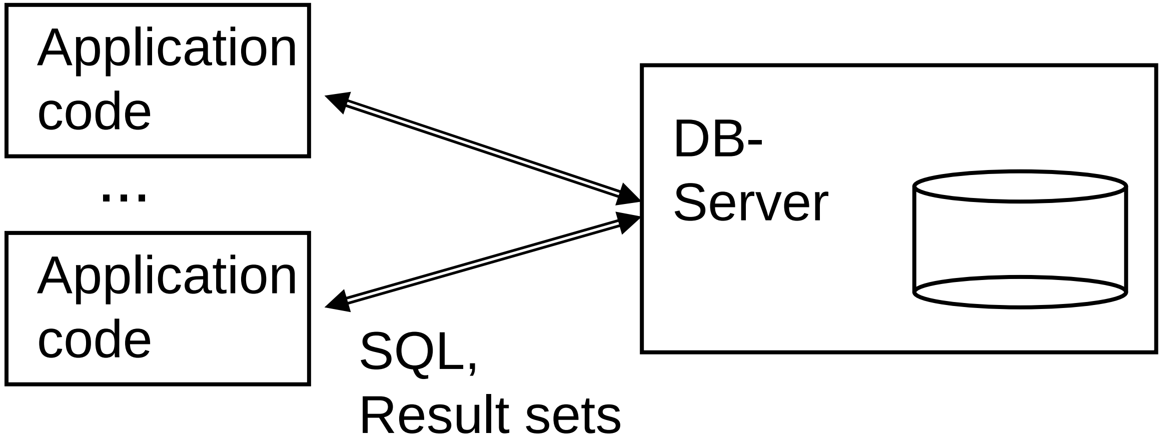
- **Business logic:** the steps which have to be made in order to process a user query.
  e.g. "go to check out" in an Internet shop is implemented by several steps, most of them access the DB:
  User logged in? if not..., perform stock keeping operations, prepare invoice, charge client, ...

- **Two tier or Three tier:** ~ business logic separated from user interaction as well as data access?

Architectures

Client server model

- **Business logic** sits in **application program**
- Runs on a machine different from database server
- Interaction by means of SQL queries, inserts, updates



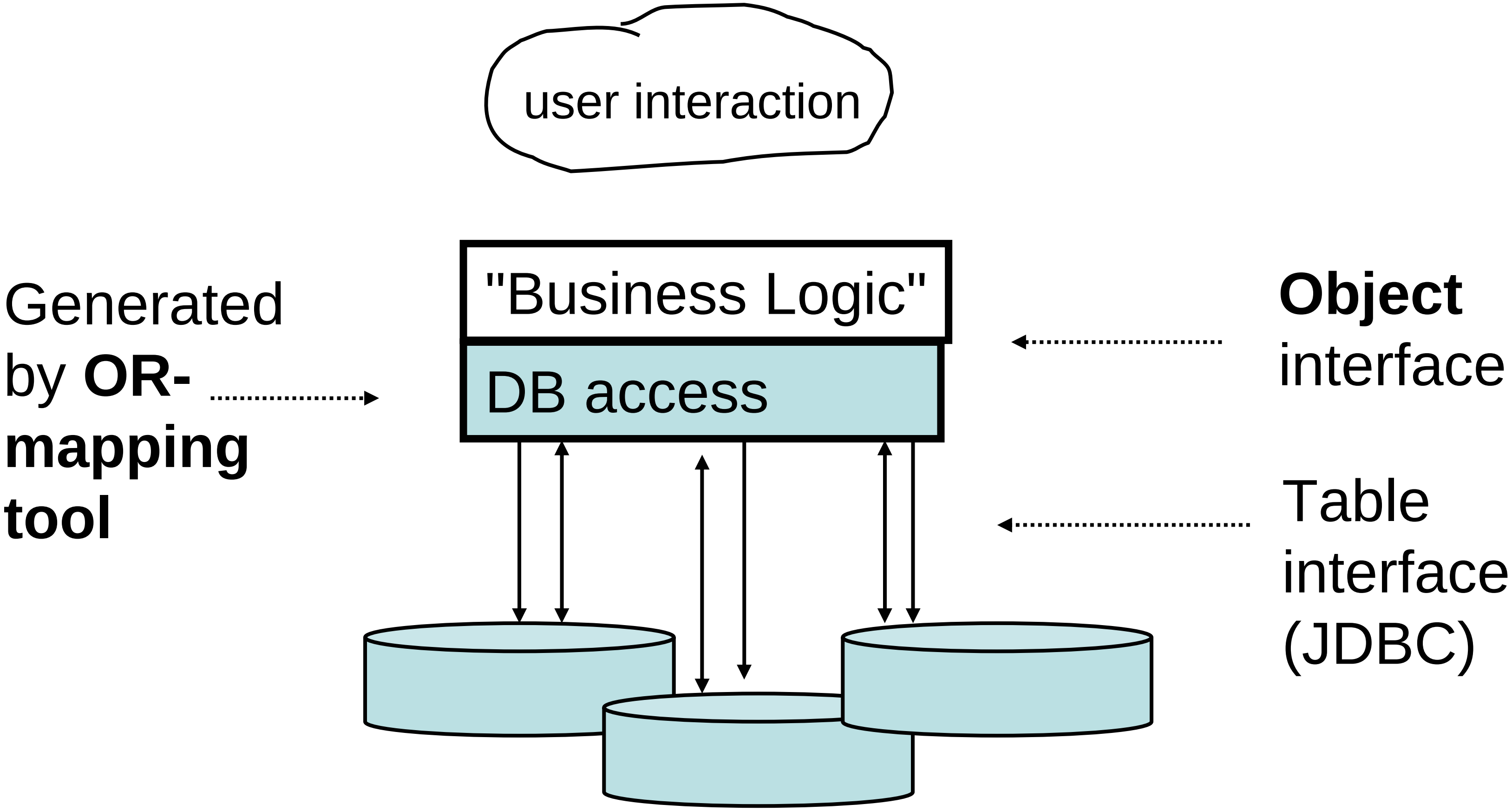
User interaction: web browser or integrated (e.g. Swing)

Client server example

```
class JdbcTest {
public static void main (String args []) throws SQLException {
// Load driver
DriverManager.registerDriver (new oracle.jdbc.OracleDriver());
// Connect to the local database
Connection conn =
  DriverManager.getConnection ("jdbc:oracle:thin:@myhost:1521:orcl",
"hr", "hr");
// Query the employee names
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery ("SELECT last_name FROM
   employees");
// Print the name out
 while (rset.next ())
    System.out.println (rset.getString (1));
// Close the result set, statement, and the connection
rset.close();
stmt.close();
conn.close();
}
}
```

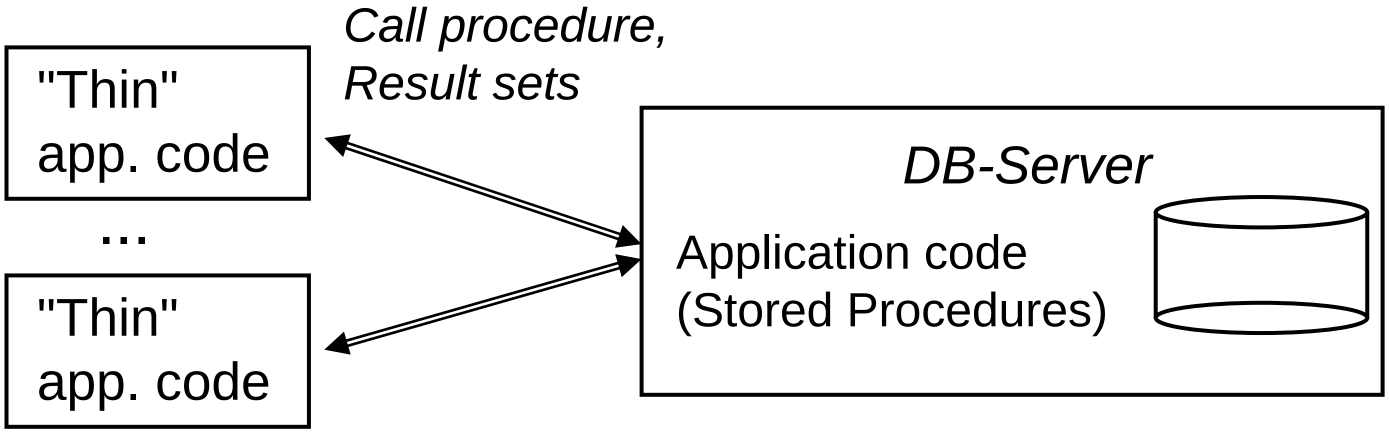
Persistence abstraction mechanisms

Object oriented programming model with persistence abstraction hides SQL database access



Server side application logic

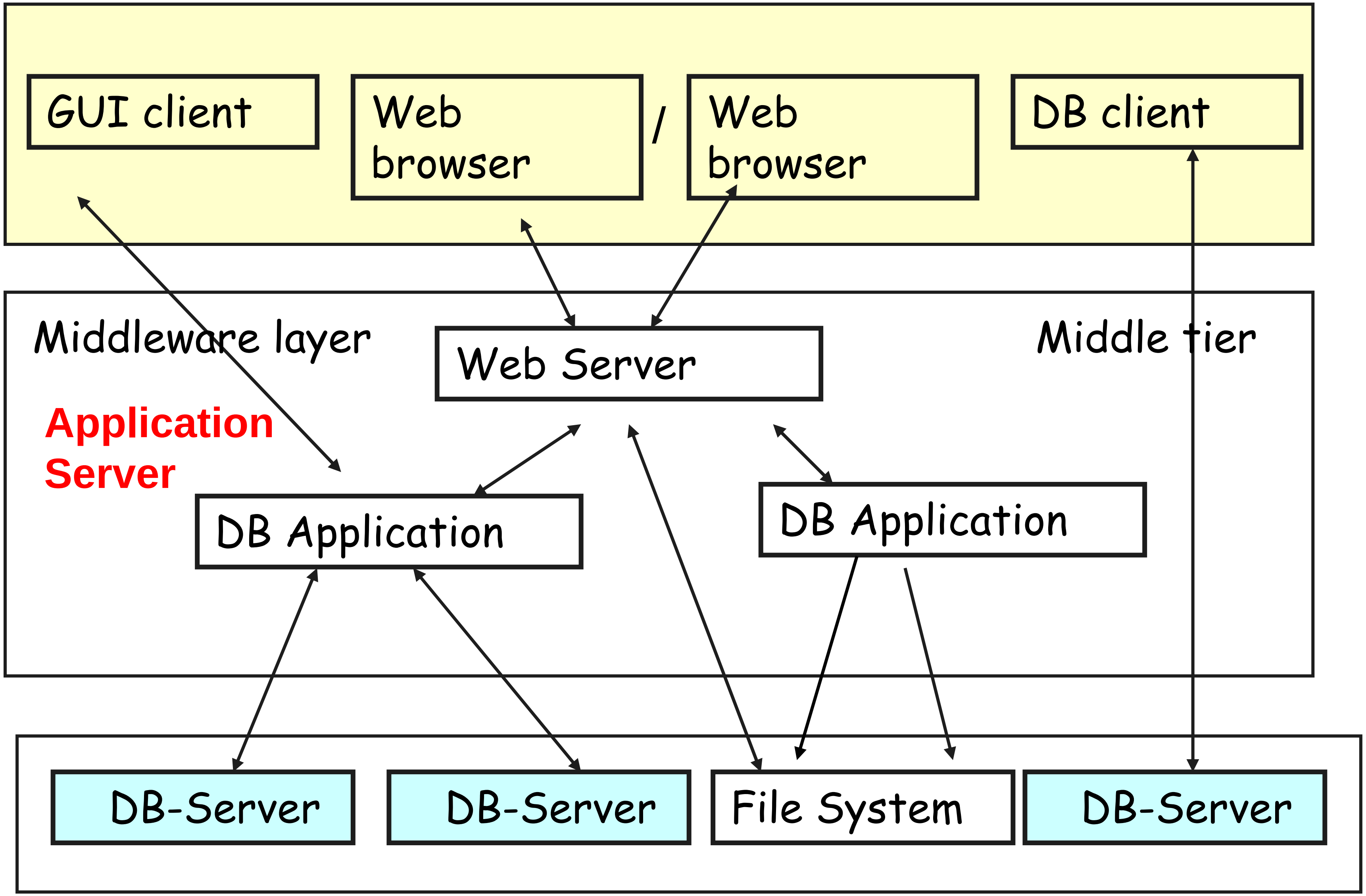
- Business logic in **stored procedures**


"Thin" app. code
...
"Thin" app. code
Call procedure, Result sets
DB-Server
Application code (Stored Procedures)

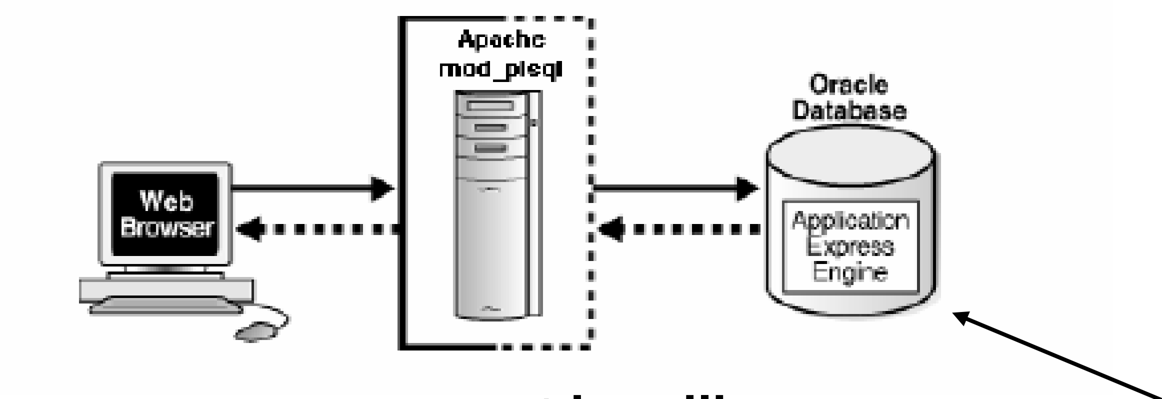
Thin clients

- Stored procedures written in **DB specific host language** e.g. PL/SQL, PL/pgSQL based on SQL/PSM standard
- **Programming language** like C, C++, Java,

Multi tier architecture

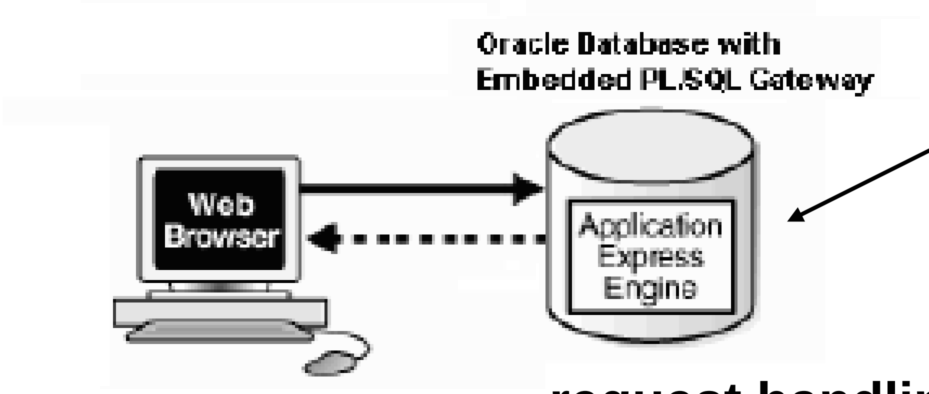


Server side architectures



request handling in web server

Basically stored procedures



request handling in DB server

Pros and Cons

Server based code:

+ **performance**
+ communication efficiency
+ Database servers provide (most of) the functionality

Multi tier architecture

+ **scalability**
+ interoperability of autonomous systems
+ secure and reliable transport of request / reply messages
+ Better workflow support

But base technologies are basically the same
in both architectures...

Base technologies

... to come:

- **Database script languages** (like **PL/pgSQL**)

  also used for trigger programming
- **Stored procedures using Java, C** or alike
- **Embedding SQL** into programming languages

  call level interface e.g. JDBC

  integration in PL e.g. Embedded SQL ESQL/C,
  java integration: SQLJ
- **Object relational mapping**: hiding data access and persistence from application code.

8.3 Stored procedures

Server extension by user defined functions

SQL based: **PL/SQL** (Oracle), **PL/pgSQL**

- adds control structures to SQL
- easy way to define complex functions on the DB

Programming language based

C, Java, ...,Perl, Python, Tcl for Postgres

Any Programming language suitable in principle

SQL standards

DB-Script languages

- Based on **SQL/PSM** ("persistent stored modules") standard
- Only propriatary implementations: PL/SQL (Oracle), PL/pgSQL (Postgres), Transact-SQL (Micorsoft), SQL procedure language (IBM)
- But conceptually similar

Programming language based

- SQL/OLB (object language binding)
- SQL/JRT (SQL routines and types using the Java language)
- SQL/CLI (SQL call level interface): How to call SQL from Programming language.

DB script languages basics: Blocks

Syntax

```
[DECLARE
  /* Declarative section: variables, types, and local subprograms. */ ]
 BEGIN
  /* Executable section: procedural and SQL statements go here. */
  /* This is the only section of the block that is required. */
[EXCEPTION
  /* Exception handling section: error handling statements go here. */   ]
END;
```

Block: **Scope** as in programming languages,
nesting allowed.

Usage

- Blocks used for **direct excecution** (e.g. SQL +)
  (only for testing and some administrative tasks)

- Used within programs. e.g. C

```
EXEC SQL EXECUTE
     < Block >
```

- Definition of independent functions / functions

```
CREATE PROCDURE … (…) IS
```

- For definition of **triggers**

- Inside object / type declarations

```
CREATE TYPE BODY
```

Type definitions: see below

Declarations

Standard declarations

```
DECLARE
 price       NUMBER;
 prodName    VARCHAR(20);
```

All variables have to be declared, all SQL types allowed.

Use table types

```
DECLARE
 prodName Product.name%TYPE;
```

table

column

Use row type

```
DECLARE   productTuple Product%ROWTYPE;
```

This is a **record** type

Record types

Example

PL/SQL syntax

```
DECLARE countryRec Country%ROWTYPE;
BEGIN
 SELECT * INTO countryRec FROM Country WHERE CODE='D';
 dbms_output.PUT_LINE('Name: ' || countryRec.name);
END;
```

Library function (Oracle)

- May be executed from the command line
- Works only with exactly one result row
- How to iterate over result sets?

PL/SQL Control flow

```
CREATE TABLE TNumb
   (x NUMBER, y NUMBER);

DECLARE
 i NUMBER := 1;
BEGIN
LOOP
 INSERT INTO T1 VALUES(i,i+1);
 i := i+1;
EXIT WHEN i>100;
END LOOP;
END;
```

Only SQL/DML within block

Similar : `WHILE (<condition>) LOOP ... END LOOP`

`FOR <var> IN <start>..<finish> LOOP...END LOOP`

see Manual

PL/SQL Insertion in FOR loop

```
CREATE TABLE TestNormal (empno number(10), ename
  varchar2(30), sal number(10));

BEGIN
FOR i in 1..1000000
LOOP
   INSERT INTO Test_normal
    VALUES (i, dbms_random.string('U',80),
        dbms_random.value(1000,7000));
   IF mod(i, 10000) = 0 THEN
   COMMIT;
   END IF;
END LOOP;
END;
```

Library function

Transaction commit: inserted data stored in DB now.

All or nothing semantics.

Result sets

Problem: how to process result set of unkown cardinality?

```
DECLARE countryRec Country%ROWTYPE;
BEGIN
 SELECT * INTO countryRec FROM Country WHERE CODE='D%';
 dbms_output.PUT_LINE('Name: ' || countryRec.name);
END;
```

...does not work – more than one result record expected.

Needed: a kind of **pointer to result set records**, which allows to **iterate through** the **result set.**

Result set: example

```
DECLARE
 CURSOR ctry IS
     SELECT *  FROM Country WHERE CODE LIKE 'D%';
 countryRec Country%ROWTYPE;
BEGIN
 OPEN ctry;
 LOOP
  FETCH ctry INTO countryRec;
  EXIT WHEN ctry%NOTFOUND;
  dbms_output.PUT_LINE
   ('Name: ' || countryRec.name || ', Popul: '||
    countryRec.population);
 END LOOP;
 CLOSE ctry;
END;
```

Cursor, internal object, not a variable

has few operations:
`OPEN, CLOSE, FETCH`

and attributes:
`%NOTFOUND`,
%**`OPEN`**,
%**`ROWCOUNT`** et al

Cursor (*)

Def: A **cursor** is an abstraction of a result set for a *particular SQL statement* with operations: OPEN, FETCH, CLOSE and attributes %ROWCOUNT, %FOUND, %NOTFOUND

- **Explicit** cursors have to be defined for SQL statements with more than one result record
- **Implicit cursors** are defined for every SQL statement

```
BEGIN
DELETE FROM TNUMB WHERE x > 50;
DBMS_OUTPUT.PUT_LINE('Deleted rows: ' || SQL%ROWCOUNT);
END;
```

(*) Important concept for embedding SQL in host (programming) languages, typically more operations, see JDBC below

Cursors and FOR loops

```
DECLARE
 CURSOR ctry IS
    SELECT *  FROM Country WHERE CODE LIKE 'C%';
 row# int;
BEGIN
FOR resRecord IN ctry LOOP
 row# :=ctry%ROWCOUNT;
  dbms_output.PUT_LINE
     ('Name: ' || resRecord.name ||
      ', Popul: '|| resRecord.population);
 END LOOP;
 dbms_output.PUT_LINE('Number of countries: ' || row#);
END;
```

LOOP is part of FOR loop on result set of implicit cursor.

- Implicit: open, close, record variable of result record.
- Cursor closed at END LOOP, *no attributes defined after that point.*

Collection variables

```
DECLARE
  TYPE largeCtry IS RECORD (
    name country.name%TYPE,
    capital country.capital%TYPE);
  TYPE largeCtryTab  IS TABLE OF largeCtry;
  lTab largeCtryTab;
  i int;
BEGIN
 SELECT name, capital BULK COLLECT INTO lTab
 FROM country WHERE population >= 100000000;

 FOR i IN 1..lTab.LAST LOOP
  dbms_output.PUT_LINE
     ('Name: ' || lTab(i).name || ', capital: '||
   lTab(i).capital);
 END LOOP;
END;
```

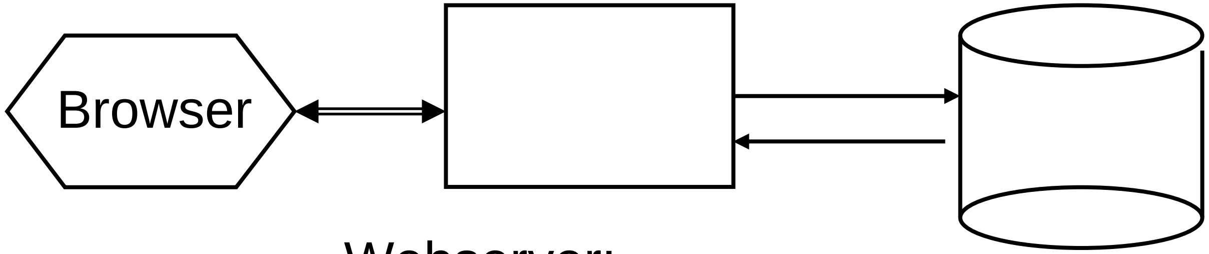
TABLE variables allow for manipulation of sets within a block

Bulk load from DB or individual assignement

Set operations in DB usually preferrable

8.4 Functions and procedures

Recall...



Webserver:

- interpret request
- call stored **procedure**
- return html

Database
with business
logic as
stored procedures

Needed: **procedures** and **functions** , not just
anonymous blocks

- Major syntactic (and some semantic) differences between PL/SQL and PL/pgSQL
- e.g. no procedure in PL/pgSQL but **`FUNCTION RETURNS VOID`**

PL/SQL procedures

```
CREATE PROCEDURE addtuple2 ( x IN T2.a%TYPE,
                             y IN T2.b%TYPE)
AS
 i NUMBER = dbms_random.value(1000,7000)
 -- here go declarations
BEGIN
      INSERT INTO T2(k NUMBER,a, b)
             VALUES(i, x, y);
END addtuple2;
```

No DECLARE (!)

Parameter passing like in ADA:

- call by value (**IN**),
- call by result (**OUT**),
- call by value-result (**INOUT**)

Why no call by reference??

Functions in PL/SQL

```
CREATE FUNCTION CountryCity(cname IN VARCHAR)
RETURNS int
IS
 CURSOR ctry IS
    SELECT *  FROM Country WHERE CODE LIKE cname||'%';
 row# int;
BEGIN
FOR resRecord IN ctry LOOP
 row# :=ctry%ROWCOUNT;
  dbms_output.PUT_LINE
     ('Name: ' || resRecord.name ||
      ', Capital: '|| resRecord.capital);
 END LOOP;
 RETURN (row#);
END;
```

Calling functions / procedures

- Embedded in **host language** like C, Java similar to execution of plain SQL → below
- Big difference: no result set, but usage of INOUT, OUT parameters and function values
- Inside PL/SQL block

```
BEGIN
  dbms_output.Put_Line('Number of countries: ' ||
TO_CHAR(CountryCity('G')));
  END;
```

- Postgres: Server Programming interface (SPI)

Packages

PL/SQL packages:

define **API and its implementation** for related functions and procedures

```
CREATE PACKAGE MyMondial AS
  TYPE myCity City%ROWTYPE;
  Cursor myC RETURNS myCity;
  FUNCTION BigCites(countryName VARCHAR) RETURN NUMBER;
  PROCEDURE NewCityInsert(newC myCity);
END MyMondial;
```

The API for this package

```
CREATE PACKAGE BODY MyMondial AS
  myVar NUMBER; -- local to package!
  CURSOR myC AS SELECT * FROM City WHERE.. –-full def.
  FUNCTION BigCities(...)AS ... –- full definition
  PROCEDURE NewCityInsert(newC myCity) AS...; --full def.
BEGIN ...   -- initializations
END MyMondial
```

Implementation

PL/SQL: etc

Exception handling

```
EXCEPTION
 WHEN <exceptionname> [OR…]
    THEN <SQL / PL/SQL – statement sequence>;
 WHEN OTHERS
    THEN <SQL /PL/SQL – statement sequence>
```

- Flexible concept comparable with Java exceptions.
- Different semantics for special situations.
  (see manual)

Realistic PL/SQL (Oracle) example

```
-- very simple purchase transaction
CREATE PROCEDURE Purchase() AS
    qty_on_hand  NUMBER(5);
BEGIN
    SELECT quantity INTO qty_on_hand FROM inventory
        WHERE product = 'TENNIS RACKET'  --
        FOR UPDATE OF quantity;

    IF qty_on_hand > 0 THEN  -- check quantity
        UPDATE inventory SET quantity = quantity - 1
            WHERE product = 'TENNIS RACKET';
        INSERT INTO purchase_record
            VALUES ('Tennis racket purchased', SYSDATE);
    ELSE
        INSERT INTO purchase_record
            VALUES ('Out of tennis rackets', SYSDATE);
    END IF;
    COMMIT;
END;
/
```

PL/pgSQL in a nutshell

Example

```
CREATE FUNCTION foo (acc integer, amount numeric) RETURNS
   numeric AS
  $B$ UPDATE bank SET balance = balance - amount
       WHERE accountno = acc;
     SELECT balance FROM bank WHERE accountno = acc;
  $B$ LANGUAGE SQL;
```

$ quoting of PG

- Many SQL-statements in one call: performance gain
- value returned: first row of last query result
- Compound result type and table valued functions allowed

⇒ Table valued function in FROM clause

SQL based functions

Table result types

```
CREATE FUNCTION getfoo(integer) RETURNS SETOF movie AS $$
   SELECT * FROM movie
    WHERE m_id = $1;
$$ LANGUAGE SQL;
```

placeholder for parameters

```
SELECT title, director FROM getfoo(93) AS m1;
```

Alias for returned table value

PL/pgSQL in a nutshell

Example

```
CREATE OR REPLACE FUNCTION rand (hi integer,low int4)
 RETURNS integer AS
 $BODY$
  -- no DECLARE
  BEGIN
    RETURN low + ceil((hi-low) *  random());
  END;
 $BODY$
 LANGUAGE 'plpgsql' VOLATILE;
```

Here go the variable declarations

Standard functions: `random()` returns uniformly distributed values 0<= v <= 1.0

$-quote, useful for string literals

Function may not return the same value for same argument: hint for optimization

PL/pgSQL in a nutshell

```
CREATE OR REPLACE FUNCTION video.randtab(count integer,
                                low integer, hi integer)
RETURNS integer AS
$BODY$
   DECLARE c INTEGER :=0;
           r INTEGER;
   BEGIN
     CREATE TABLE randomTable (numb integer, randVal
                                integer);
     FOR i IN 1..count
      LOOP
      INSERT INTO randomTable VALUES(i, rand(low,hi));
     END LOOP;
     RETURN (SELECT MAX(numb) FROM randomTable);
   END;
  $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

variable declarations

side effects!

PL/pgSQL in a nutshell

Evaluation of functions

Within a select statement:

```
SELECT randtab(100,0,9)
```

Without result value

```
PERFORM my_function(args)
```

EXECUTE query plan

```
EXECUTE PROCEDURE emp_stamp();
```

Note: Functions may have side effects!

No (pretty) PRINT facilities

workarounds: `SELECT 'This is my heading'`

- put PLSQL-call into shell script
- use Programming language for I/O

8.5 Triggers

Triggers: Event – Condition – Action rules

Event: **`Update, insert, delete`** (basically)

Condition: **`WHEN`** < some conditon on table>

Action: some operation (expressed as DML, DB- Script language expression, C, Java,…)

Triggers make data base systems **<u>pro-active</u>** compared to **<u>re-</u>active (and interactive)**

Triggers: simple example

Basic Functionality

```
CREATE TRIGGER myTrigger
  BEFORE [AFTER]  event
  ON TABLE myTable FOR EACH ROW  { | STATEMENT}
  EXECUTE PROCEDURE myFunction(myArgs);
```

event: UPDATE, INSERT, DELETE

Semantics

Execute the function **after each** event

once for each row changed **or once per statement**

e.g. per statement: write log-record

per row: write new time-stamp

Anatomy of a trigger (Oracle)

```
CREATE OR REPLACE TRIGGER movie_DVD_Trigger
INSTEAD OF INSERT ON T_M
FOR EACH ROW

DECLARE m_row NUMBER;
-- local variable
BEGIN
 SELECT COUNT(*) INTO m_row
 FROM Movie
 WHERE m_id = :NEW.mid;

 IF m_row = 0
 THEN RAISE_APPLICATION_ERROR(-20300, 'Movie does not exist');
 ELSE INSERT INTO DVD (DVD_id, m_id) VALUES (:NEW.DVD_id,
   :NEW.mid);
 END IF;
End;
```

Semantics: trigger for each row affected (not only once per excecuted statement)

Action (here: PL/SQL)

```
CREATE view T_M
AS SELECT m.m_Id AS mid, DVD_id, title
...
```

Using an INSTEAD OF TRIGGER

Without the trigger:

```
Insert into T_M (mid, DVD_id) VALUES(93,14);
   *
FEHLER in Zeile 1:
ORA-01779: Kann keine Spalte, die einer Basistabelle zugeordnet
   wird, verändern
```

Using the INSTEAD OF TRIGGER

```
Insert into T_M (mid, DVD_id) VALUES(93,14)
1 Zeile eingefügt
```

```
Insert into T_M (mid, DVD_id) VALUES(99,14)
            *
FEHLER in Zeile 1:
ORA-20300: Movie does not exist
ORA-06512: in "VIDEODB.MOVIE_DVD_TRIGGER", Zeile 8
ORA-04088: Fehler bei der Ausführung von Trigger
   'VIDEODB.MOVIE_DVD_TRIGGER'
```

Triggers...

... are a powerful DB programming concept

Allow complex integrity constraints

Used in most real-life database applications

Sometimes dangerous:

```
CREATE TRIGGER myTrigger1
  BEFORE INSERT
  ON TABLE myTable1  EXCECUTE myfct (...)
            -- inserts some record into myTable2

    CREATE TRIGGER myTrigger2
  BEFORE INSERT
  ON TABLE myTable2  EXCECUTE myfct (...)
            -- inserts some record into myTable1
```

Cycle!

8.6 SQL3: Abstract data types

"ADT is a data type **defined by the operations** allowed on its values"

```
CREATE TYPE  <name> (
        <list of component attributes>
        <declaration of EQUAL, LESS>
        < declaration of more methods> )
```

supported only by a few DBS

ADT equivalent to 'object type' (Oracle)

... or functions may be defined stand-alone (PG)

Functions, methods, procedures

Method interface in an object type definition
(Oracle flavor)

```
CREATE TYPE LineType AS OBJECT
  ( end1 PointType,
    end2 PointType,
    MEMBER FUNCTION length(scale IN NUMBER) RETURN
                                            NUMBER,

    PRAGMA RESTRICT_REFERENCES(length, WNDS));

CREATE TABLE Lines ( lineID INT, line LineType );
```

Predicates defined over functions

```
 SELECT lineID, k.length (1.0) FROM Lines k
  WHERE k.length(1.0) > 8.0
```

Defining methods (Oracle)

Implementation of a method signature*

```
CREATE TYPE BODY LineType AS
  MEMBER FUNCTION length(scale NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN scale * SQRT((SELF.end1.x-
         SELF.end2.x)*(SELF.end1.x-SELF.end2.x) +
        (SELF.end1.y-SELF.end2.y)*(SELF.end1.y-
        SELF.end2.y) );
  END;
 END;
```

Methods may be defined in Java or PL/SQL (Oracle)

Functions: independent of types, no SELF attribute

*compare: java interface vs. class

see: Ullman, J.: Object-Relational Features of Oracle
http://www-db.stanford.edu/~ullman/fcdb/oracle/or-objects.html

Summary

- Extensions of relational model popular
- SQL 3 keeps extensions under control – somehow
- Object-relational extensions more important than object oriented database systems
- Extensions basically are:

  structured types and set types

  functions, written in a db script language or
  some programming language

  active elements: triggers (SQL 3) , rules (only PGres)