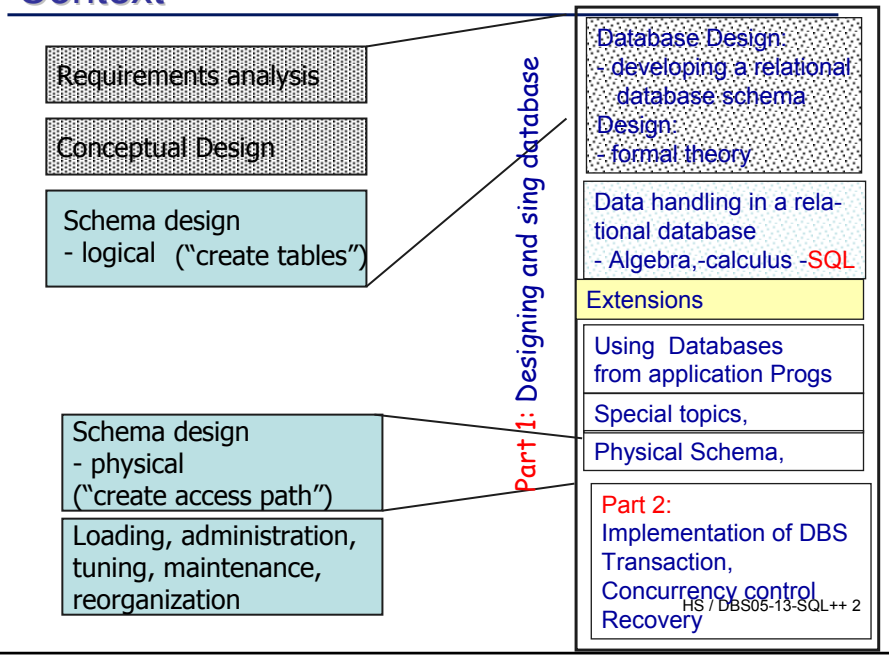
10. Extending the Relational Model: - SQL 99 –

see Kemper/Eickler chap. 14, Elmasri chap. 6.8, O'Neill: Chap. 4, Postgres manual

Context



10.1 Motivation Objects

- Missing from relational model
  - Structured attributes
    - Compound records
    - Vectors, sequences
    - Sets
    - tables
    - Type constructors
  - Inheritance between relations
    - ER generalization and mapping to relations useful on the design level, not as relations
  - Operations (methods, functions)
- Principal solutions
  - Make object oriented language persistent
  - Enhance relational model carefully with object concepts

Motivation OO Languages

- Persistent OO Programming Language
  - Ideally no difference between persistent and non persistent objects form a programmer's point of view "Seamless integration"
  - Use OO Modeling and design methodology
  - Issues
    - Query language or pointer chasing?
      First generation ODBS did not have a query language
    - Concurrency and transactional support?
    - Schema?
    - What about non-object oriented languages?
  - Gives up the "tabular data abstraction"
    - Neutral Data format
    - Powerful operations
  - Object Oriented Database Systems
    Object Store, Poet, O2, GemStone, Orion, …

Motivation Objects and Relations

- Object Relational Systems
  - Principles:
    - Keep the goodies of RDB
    - Enhance relational systems by constructed types, inheritance, methods
  - Supported by SQL-3 standard
  - Issues
    - Technologically outdated solutions stabilized
      e.g. type system, baroque constructs of SQL
    - Add-on usually worse than completely new approach
  - Important aspect: Save investment into your software
    - DBS, application software
      Example: 50.000 programs to change when moving from hierarchical DBS (IMS) to RDBS (~1988, German car manufacturer)

Overview: SQL-3 extensions

- Collection types
  - SET, Multiset, List constructors for primitive and constructed types
    e.g.
```
Phones SET (VARCHAR(20))
Polygon LIST (Point_T)
```
- Composite types / objects
  Records, objects
- Functions / methods
- Active elements: Triggers

Big differences between 'object relational' products:
Oracle, DB2, (Informix), Postgres

10.2 Collection types: arrays

- SQL -3 standard:

```
CREATE TABLE phone_book (
     pid        INTEGER,
     home       VARCHAR(15)   ARRAY[5]
     mobile     VARCHAR (15) ARRAY[3])
```

- Postgres arrays
  - additional syntax:
    <attribute> <basetype> [<n>]
  - more than one dimension

```
CREATE TABLE chessBord (
   board      INTEGER [8][8]
```


Collection types: arrays

- Oracle VARARRAYs:
  - special case of general type extension

```
CREATE TYPE Extension_T as VARRAY(4) OF Int;
CREATE OR REPLACE TYPE Address_T AS OBJECT (
  zipcode      CHAR(6),
  ciy          VARCHAR(20),
  street       VARCHAR(25),
  no           CHAR(5));

CREATE TABLE phone_book (
  name         VARCHAR(30),
  firstName    VARCHAR(20),
  addr         Address_T,
  phones       Extension_T);
```

Type definition, see below

Collection types: arrays

- How to update / select arrays?

  Postgres syntax for array constants:

  e.g `('a','b','d')  , ((1,2),(1,2,3,4))`

```
INSERT INTO phone_book VALUES
 (11, ('+49-30-4836721', '4398722'),() )
```

  Array constructor: `ARRAY [1,3,5,7,9]`

  also allowed.

Collection types: arrays

- Selection

```
SELECT pid FROM phone_book
 WHERE home[1]  = '4711348'
```

  Questionable! One of the phone numbers should satisfy the predicate, not just the first.

  Postgres solution: IN, ANY/SOME, ALL set comparisons

```
SELECT pid FROM   phone_book
  WHERE   '4711348'  IN home
```

Collection types: arrays

- Arrays are not sets
  ⇒ separate tables are a better design decision for sets
- Implementation issues:
  Postgres stores array values within the rows
  ⇒ Problems with large number of values
  (e.g. `measurement REAL [1000]`)
- Completely different handling of arrays in Oracle

Collection types: arrays (Oracle)

– VARRAY defined

```
INSERT INTO PhoneBook VALUES(
  'Abel', 'Hendrik', ADDRESS_T('12347',
 'Berlin', 'Takustr.','9'),
  Extension_T(2347, 1139));

SELECT b.name, b.addr.street, b.phones
FROM PhoneBook b
WHERE 2347 IN (SELECT * FROM TABLE(b.phones));
NAME  ADDR.STREET      PHONES
--------------------------------
Abel  Takustr.        EXTENSION_T(2347, 1139)
```

Cast of a varray to a table

– No way to address by position
e.g. phones[2]

10.3 Creating new types

- Definition of composite types

```
CREATE TYPE complex AS
 ( r double precision, i double precision );
```

Nothing but a definition of the type of a structured value ("record")

Used as a column type in Postgres

```
CREATE TABLE Foo (
  n VARCHAR(5),
  roots complex [2]
)
```

Access to components of a type:

```
SELECT n, (roots).x FROM Foo WHERE...
```

note '(...)'

Nested tables (SQL-3): Oracle flavor

- Nested table
  - Restriction: only one level of nesting

```
CREATE TYPE Polygon_T AS TABLE OF Point_T;

CREATE TABLE Polygons (
   pid       CHAR(5),
   points    Polygon_T)
   NESTED TABLE points STORE AS PointsTab;
```

Stored outside table `Polygons`

Nested tables (SQL-3): Oracle flavor

– Querying nested tables using inner tables

```
SELECT t.pid FROM Polygons t
WHERE EXISTS
(SELECT ss.x FROM
    TABLE(SELECT points
          FROM Polygons t2
          WHERE t2.pid =t.pid
             ) ss
     WHERE ss.x = ss.y)
 AND pid <> 'squ01';
```

Select one row in polygons and select a table value (points)

Additional outer qualification predicate

Inner qualification on table value

Object Relational Concepts Object types

- SQL-99 objects
  - Row types
    A row may be an object which has attributes and can be referenced
  - Column types
    an (A)DT which may be the type of an attribute

```
CREATE TYPE AdressType AS OBJECT (
   city      VARCHAR(30),
   zipCode   VARCHAR(15),
   street    VARCHAR (30),
   number    VARCHAR(5),
   country   VARCHAR(15)
   );
   /
```

Object Relational Concepts — Object types

- Object Types, Object Tables and REFs (SQL-3, Oracle)
  - REF types have values, can be seen by user as opposed to OO languages

```
CREATE TYPE MovieType AS OBJECT (
    mId          Int   ,
    title        VARCHAR(30),
    director     VARCHAR(25),
    year   date
); /
```

Object table

```
SQL> CREATE TABLE MovieObjects OF MovieType;
```

```
CREATE TABLE MovieTape (
    Id          Int,
    movie       REF MovieType,
    since       date,
    back        date
);
```

Table with object reference

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 );
```

Parameter type: in, out, in out

- Predicates defined over functions

```
SELECT lineID, l.length (1.0) FROM Lines l
WHERE l.length(1.0) > 10.0
```


Defining methods (Oracle)

- Implementation of a method signatur*

```
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;
```

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

*compare: java interface vs. class

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

Type definitions in PostgreSQL: composite

Completely different concept

A. composite types

```
CREATE TYPE point AS ( r double precision, i double
   precision );

CREATE TABLE polygons (
         pid       CHAR(5),
         ppoints    point[200])

SELECT  ppoints[1] AS startPoint
 FROM  polygons WHERE pid = 'myPoly'
```

No object types:
- no REF,
- no inheritance,
- no methods

Postgres offers geometric types **`like point, line, ...`**

They have been defined using the "user defined type" concept

Type definitions in PostgreSQL: user defined

B. User defined types

... are scalar types

- not composite from SQL point of view

- Defined as a C structure (typically)
- Needed: input and output functions
- no general method interface

Example: complex numbers*

```
typedef struct Complex
     { double x; double y; } Complex;
```

* see PG manual

Type definitions in PostgreSQL

Input function for user defined data type 'complex':

```
complex_in(PG_FUNCTION_ARGS)
{ char *str = PG_GETARG_CSTRING(0);
 double x, y;
 Complex *result;
 if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
    /* parser for input and error handling....
 result = (Complex *) palloc(sizeof(Complex));
 result->x = x; result->y = y;
 PG_RETURN_POINTER(result);
}
```

Registering as a PG function:

```
CREATE FUNCTION complex_in(cstring) RETURNS complex
  AS 'myComplex.c' LANGUAGE C IMMUTABLE STRICT;
```


Type definitions in PostgreSQL

Defining the scalar type

```
CREATE TYPE complex
 ( internallength = 16,
    INPUT = complex_in,
    OUTPUT = complex_out,    /* output fct
    RECEIVE = complex_recv, /* binary input fct(option.)
    SEND = complex_send,     /* binary output
    ALIGNEMENT = double );
```

Using the scalar type

```
CREATE TABLE foo AS
  (...,
    val complex,
    ...
  )
```

Type definitions in PostgreSQL

- Example: Document type

```
CREATE TYPE document (
  INPUT = lo_filein,
  OUTPUT = lo_fileout,
  INTERNALLENGTH = VARIABLE );

CREATE TABLE myDocuments ( id integer, obj document );
```

- Not very interesting...
  .... without operators / functions
- Functions may be defined independently

```
CREATE FUNCTION concat_docs(doc, doc)
  RETURNS doc AS 'concat_docs',
  LANGUAGE C;
```


10.4 Functions ('stored procedures')

Server extension by user defined functions

- defined independent from type / object type
- either
  - SQL based: SQL, 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
- Defined in the SQL 3 standard,
- (Some) differences in most systems

SQL based function

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(varchar(30)) RETURNS SETOF
  movie AS $$ SELECT * FROM movie
   WHERE m_id = $1;
$$ LANGUAGE SQL;

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

placeholder for parameters

Alias for returned table value

PL/pgSQL in a nutshell

- First example

During testing no DROP necessary

```
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

block

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

$-quote, useful for string literals

Function does 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

```
CREATE OR REPLACE FUNCTION video.actorBD(
  video.actor.stage_name%TYPE)
  RETURNS DATE AS
$BODY$
  DECLARE myRec video.actor%ROWTYPE;
  BEGIN
   SELECT INTO myRec * FROM actor WHERE stage_name = $1;
   IF NOT FOUND THEN
      RAISE NOTICE 'no actor with this stage name';
  RETURN NULL
   ELSE RETURN myRec.birth_date;
   END IF;
  END;
$BODY$
  LANGUAGE 'plpgsql' STABLE;
```

only arg type type is column type of a table

Exception handling

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!

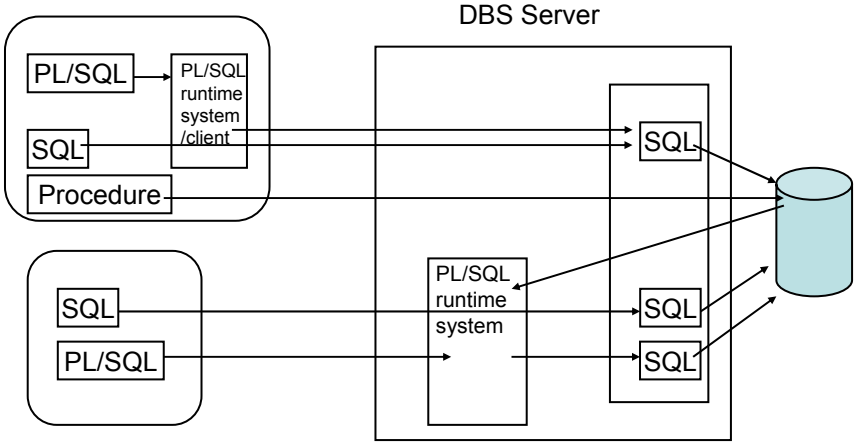
Realistic PLSQL example

```
-- very simple purchase transaction
DECLARE
    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;
/
```


Stored procedures: system architecture

Runtime system of PL/SQL (example, Oracle)



Fat / thin client

10.5 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 proactive compared to reactive

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_tape_Trigger
INSTEAD OF INSERT ON T_M
FOR EACH ROW
```

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

Action (here: PL/SQL)

```
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 Tape (t_id, m_id) VALUES (:NEW.t_id,
   :NEW.mid);
 END IF;
End;
```


Using an INSTEAD OP TRIGGER

Without the trigger:

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

Insert into T_M (mid, t_ID) VALUES(99,14)
            *
FEHLER in Zeile 1:
ORA-20300: Movie does not exist
ORA-06512: in "VIDEODB.MOVIE_TAPE_TRIGGER", Zeile 8
ORA-04088: Fehler bei der Ausführung von Trigger
   'VIDEODB.MOVIE_TAPE_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  EXCEUTE myfct (...)
          -- inserts some record into myTable2

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

Cycle!

Summary

- Extensions of relational model very popular
- SQL 3 keeps extensions under control – somehow
- Object-relational extensions more important than object oriented database systems
- Extensions basically are:
  - stctured types and set types
  - functions, written in a db script language or some programming language
  - active elements: triggers (SQL 3) , rules (only PGres)