

7. SQL – Data Handling

7.1 The query language SQL

- Search predicates
- Arithmetic expressions and functions in predicates
- Different kinds of join
- Output layout

7.2 Advanced SQL

- Subselects and Correlated subqueries
- Quantified expressions, SOME, ANY
- Grouping and Aggregation
- Transitive closure

7.3 Update, Deletion, Insertion and bulk load*

Lit.: Melton / Simon, Understanding SQP 1999, chap. 2,5,7; Kemper / Eickler chap 4, SQL chapter in any book on DBS

(*) chap.6. Calculus Language: not discussed in class

SQL / DML: Overview

Query data

- Interactively
- Embedded in host language
important in applications, next chapter

Insert, update, delete data

7.1 The Query Language SQL

Freie Universität



Berlin

SQL is relational complete

...but many additional query concepts compared to RA

Advanced search predicates on strings

e.g., find all cities starting with “Ber”

Arithmetics in expressions,

e.g., GNP / population for all countries

Grouping and predicates over sets

e.g., total GNP of EU countries

Recursion

Simple SQL Search predicates

Defined like Boolean predicates in RA and Calculus

Some syntax extensions

- `<attribute> BETWEEN <value1> AND <value2>`
- `<attribute> IS [NOT] NULL`

```
SELECT *  
FROM Country  
WHERE population BETWEEN 50000000 AND 70000000  
AND GNP IS NOT NULL
```

Simple SQL Search expressions

Simplified OR

```
SELECT Name
FROM Country
WHERE C_ID IN ('BRD', 'D', 'DDR')
```

Equivalent to

```
<attr> = <val1> OR <attr> = <val2> OR ...
```

More general case of a **table constant**:

```
SELECT Name
FROM Country
WHERE (C_ID, capital)
      IN (('BRD', 'Bonn'),
         ('D', 'Bonn'),
         ('DDR', 'Berlin(Ost)'))
```

Boolean logic ...

.. is crucial:

```
SELECT Country.Name
FROM Country
WHERE NOT (capital != 'Berlin' OR
capital != 'Wien')
```

Result?

Row predicates with conjunctive primitives

a = <value> AND a = <differentValue> will never have a non-zero result.

Remember...

All logical differences are big differences
(Wittgenstein)

Corollar:

All logic mistakes are big mistakes



not so big

3-valued logic

NULL values

- comparison result may be "unknown" if an argument is NULL
 - "unknown" result tuples in any result set
- **comparison only by IS [NOT] NULL**

\wedge	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

or, not
?

```
(select Country
from Economy
where gdp >= 0
) intersect
(select Country
from Economy
where gdp IS NULL)
```

Result set : \emptyset

Arithmetic ,functions in search predicates and more

Extensions of RA - Arithmetic expressions

May occur in simple predicates and target list

Basically arithmetic expressions (over attributes and values) allowed when attribute names allowed.

```
SELECT c.name,  
       e.gdp/c.population AS "GDP ($) per Person"  
FROM Country c join Economy e  
  ON c.code = e.country  
WHERE (gdp*1000000/population) < 500
```

Result is NULL if any involved attribute is NULL

Simple SQLString search expressions

String expressions

Simple form of regular expressions: LIKE

LIKE patterns:

- % : any sequence of characters
- _ : exactly one character

```
SELECT Country.name , capital
FROM Country
WHERE capital LIKE '%co%'

NAME                CAPITAL
-----            -----
Russia              Moscow
...
```

Regular expressions defined in SQL99*

- <string> SIMILAR TO <pattern>
- REGEXP_Like (<attr>, <pattern>) (Oracle)

Simple SQL Functions in search expressions

Built in functions

- Expressions may contain functions
- Many arithmetical and string built-in functions
- User defined functions on user defined types (see below)

```
SELECT Country.name , capital
FROM Country
WHERE SOUNDEX(capital) = SOUNDEX
( 'Monakko' )
```

NAME	CAPITAL
-----	-----
Nicaragua	Managua
Monaco	Monaco

Simple SQL Search expressions

More expressions

Arithmetic expressions

example

Constant	7.5, 3.
[qualifier.]columnname	T.format, fname
arithExpr op arithExp	3 + 4, price - 2*discount
function (aexpr)	Sqrt(xCoord*xCoord + yCoord * yCoord)

Character and Dateexpressions

cexpr '+' cexpr	'The winner is: ' + fname
function (cexpr)	SOUNDEX ('Meier')
	UPPER (fname), LOWER(...)
	TRIM (TRAILING ' ' FROM ' Hello ')
	SUBSTRING(fname FROM 0 FOR 4)
Date value express.	SYSDATE – date_of_birth

Date-Functions

SQL **date model** based on timestamps with/out time zone,
types: `timestamp`, `date`, `time` (of day), `interval`.

```
SELECT bike_ID, year_Bought
FROM Bikes
WHERE
MONTHS_BETWEEN(SYSDATE,year_bought) > 24 ;
```

```
SELECT Bike_ID
FROM Bikes
WHERE TO_DATE('1.1.2009') > year_bought
```

Problem: **compatibility**, e.g. functions on time values

General issue:

Casting may result in not a well defined value

e.g. a time `interval` of one year and five month to seconds

(how many month with 30 | 31 | 28 days?)

Naming

User defined type components
examples: Postgres

```
CREATE TYPE Coord AS (  
  longitude NUMERIC,  
  latitude NUMERIC)
```

```
CREATE TABLE City AS (  
  name VARCHAR(..),  
  coordinates Coord,  
  ....
```

How to access components?

```
SELECT name FROM City  
WHERE Coordinates.latitude  
           = 37.5
```

Does not work:
looks like path expression

```
SELECT name FROM City  
WHERE (Coordinates).latitude = 37.5
```

Array types much more involved! See manual.

Constructing Joins

Recipe for more involved join

Example: Find countries having cities with a population of 5,000,000 and more; list also city names and Province name and population.

1. Construct a "wide table" by **joining all "relevant" tables.**
2. Apply selection and projection to this table.

```
J-Tab(...Country.code,  
      Country.name,....  
      .. Province.country,  
      .. Province.name,  
      .....  
      City.country,  -- code  
      City.province,  
      City.name,  
      City.population,  
      .....  
      )
```

SQL join

Example (cont.)

```
SELECT co.name, ci.name, r.name, r.population
FROM Country co JOIN Province r
      ON co.code = r.country
      JOIN City ci
      ON r.name = ci.province AND
      ci.country=co.code
WHERE ci.population >= 5000000;
```

Tables "relevant" for query:

1. Those containing column used in projection or selection
Example: **Country**, **City**
- 2 Those needed to link tables of type 1.
Example: **Province**

Be careful with...

.. projection.

SQL> select * from r;

A	B	C
1	2	3
2	2	4
3	3	4

SQL> select * from s;

B	D
2	4
3	3

SQL> select * from t;

D	E
4	1
4	2

SQL> SELECT r.a, s.b FROM r JOIN s ON r.c != s.b NATURAL JOIN t;

A	B
1	2
2	2
3	2
1	2
2	2
3	2

Correct result.

BUG

```
QL> SELECT r.a, s.b FROM r JOIN s ON r.c != s.b NATURAL JOIN t;
```

A	B
1	2
1	2
2	2
2	2
3	2
3	2
2	3
2	3
3	3
3	3

Elimination of column D
in table s results in **cross
join** instead of
natural join.

wrong result!

10 Zeilen ausgewählt.

SQL / DML: joins

Natural inner join

```
<tableName> NATURAL [INNER] JOIN <tableName>
```

You should know what you are doing...

Example:

```
SELECT name, name  
FROM City c NATURAL INNER JOIN Province r;
```

City (NAME, COUNTRY, PROVINCE, POPULATION..)

Province (NAME,COUNTRY, POPULATION,...)

Result?

SQL / DML: Simple queries with joins

Inner equi-join with attribute list

```
<tableName> [INNER] JOIN <tableName>  
USING <attributList>
```

Subset of common attributes



Example:

```
SELECT City.name, Province.name  
FROM City INNER JOIN Province USING (country)  
ORDER BY City.name;
```

Strange (wrong!) result...

Explain output!

Symmetry of joins or tables

Problems with **symmetric relationships**

Example: `Neighbor_of (Country1, Country2)`

Wanted: list of neighbors of Germany.

```
SELECT Country1, Country2
FROM Neighbor_of
WHERE Country1 = 'D' or Country2= 'D'
```

Solves the problem...?

COUNTRY1	COUNTRY2
..	
CS	D
D	NL
...	

CASE

```
SELECT 'D' ,  
      (CASE  
        WHEN n1.country1='D' THEN n1.country2  
        WHEN n1.country2='D' THEN n1.country1  
        -- Else for catch all, not needed here  
      END) AS "Neighbor-Country"  
FROM Neighbor_of n1  
WHERE n1.country1='D' OR n1.country2='D'
```

Simple solution in this case: set operator

```
(SELECT country2 as benachbart  
 FROM Borders  
 WHERE Country1='D')  
UNION  
(SELECT country1 as benachbart  
 FROM Borders  
 WHERE Country2='D')
```

CASE

Case expression in "target list" very useful.

```
SELECT Country.Name ,  
       GNP/population AS "GNP ($) per  
                          Person"
```

...

throws exception if `country.population = 0` (but not: `NULL`)

Can be avoided with CASE .

Outer Join

Natural outer join

```
<tableName> LEFT|RIGHT|FULL  
NATURAL [OUTER] JOIN <tableName>
```

Outer join with condition

```
<tableName> LEFT|RIGHT|FULL [OUTER] JOIN <tableName>  
ON <condition>
```

Example:

```
SELECT r.name, c.name  
FROM Province r LEFT OUTER JOIN City c  
ON (r.name = c.province AND r.C_ID = c.C_ID)  
ORDER BY r.name
```

will find and output also **Provinces** without **cities**

Simple SQL: Output

Formatting the output

Different format, even HTML or other markup can be generated in some systems

"Find title, DVD_id and format for all movies"

```
BREAK ON name
```

```
COLUMN name HEADING "Land" FORMAT A15
```

```
COLUMN capital HEADING "Hauptstadt"
```

Don't repeat identical titles
Column formatting

Aliases for columns

```
SELECT c.name, r.name  
FROM Country c JOIN encompasses e  
ON c.C_ID = e.country  
JOIN Province r USING (C_ID)  
WHERE e.continent LIKE 'Europ%'  
ORDER BY c.name ASC;
```

System dependent
This kind holds for
Oracle/**SQL+**

7.2 Advanced SQL

Subselects and correlated subqueries

Using result relations instead of constants

```
SELECT name, country
FROM City
WHERE country IN ('DDR', 'D', 'BRD');
```

```
SELECT name, province
FROM City
WHERE province IN
  (SELECT name
   FROM Province
   WHERE population > 5000000);
```

Constant list

Query dependent constants

Independent **subqueries**

Independent outer and inner SQL block

Subqueries

Find name of country, the capital of which has less inhabitants than the capital of France.

```
SELECT name, code
  FROM Country c JOIN City ci
    ON c.capital=ci.name
    AND c.province = ci.province
 WHERE ci.population < ALL SOME [ANY]
    (SELECT population
      FROM City
      WHERE name='Paris')
```

Wrong, if more than one result of subquery.
Needed: **value quantifier** – compare with **all** or is there **any**?

Correlated Subselects

Correlated Subqueries: block structure, variables accessed in subordinate nested block

Find country name, for which the capital is at the same time the capital of a Province.

```
SELECT c.name, c.code, c.capital, c.province
FROM Country c
WHERE EXISTS
  (SELECT *
   FROM Province r
   WHERE r.country = c.code AND r.name = c.province
   AND r.capital = c.capital)
ORDER by c.name
```

Subqueries can be avoided in most cases.
How in the example above?

Advanced SQL: EXISTS

NOT EXISTS extends the language

Find countries, the capital of which has a higher population than **all** its Provinces (except the Province of the capital)

Could be expressed in **predicate logic**:

Find country etc. such that **for all** Provinces (except that of the capital) the population is less than the population of the city.

Equivalent:

Find country etc such that **not exists** provinces different from the capital's province the population of which is equal or larger than the population of the country's capital.

Advanced SQL: EXISTS

```
SELECT c.name, c.code, c.capital, c.province,  
       ci.population  
FROM Country c JOIN City ci ON ci.name=c.capital  
       AND c.province = ci.name AND c.code=ci.country  
WHERE NOT EXISTS  
(SELECT *  
  FROM Province r  
   WHERE r.country = c.code AND c.province != r.name  
        AND r.population >= ci.population )  
ORDER by c.name
```

Division and EXISTS

Find countries which are members of **all** organizations, Germany is in.

Algebra expression?

```
SELECT DISTINCT country
FROM IsMember m1
WHERE NOT EXISTS (
    SELECT * FROM IsMember m2
    WHERE country = 'D' AND NOT EXISTS (
        SELECT * FROM IsMember m3
        WHERE m3.country = m1.country AND
              m3.organization = m2.organization
    )
)
```

Set operators

Find countries which belong to Europe and Asia.

~~... where continent = 'Europe' and continent = 'Asia'~~

```
SELECT country FROM encompasses  
WHERE continent = 'Europe'
```

INTERSECT

```
SELECT country FROM encompasses  
WHERE continent = 'Asia'
```

Set operators eliminate duplicates!

`<set op> ALL` does not.

Quantification and set operators

Example from above...

```
SELECT country          -- not in Oracle
FROM IsMember m
WHERE NOT EXISTS(
  (SELECT organization FROM IsMember
   WHERE country='D'
  )
  EXCEPT
  (SELECT organization FROM IsMember m1
   WHERE m.country = m1.country
  )
)
```

Table expressions

```
SELECT name FROM country c
WHERE c.code IN (
    SELECT country FROM encompasses
    WHERE continent = 'Europe'
    INTERSECT
    SELECT country FROM encompasses
    WHERE continent = 'Asia'
)
```

Avoid subqueries, even if not correlated

Table expressions

```
SELECT name FROM country c,  
  
    ( SELECT country FROM encompasses  
      WHERE continent = 'Europe'  
      INTERSECT  
      SELECT country FROM encompasses  
      WHERE continent = 'Asia'  
      ) euroAsia  
WHERE euroAsia.country = c.code
```

table expression
in from list

Aggregation and Grouping

Aggregate (or set) functions

f_A : table \rightarrow value, where A is some Subset of $\Sigma(\text{table})$

Aggregate functions are table functions, i.e. defined on tables or subsets of tables, not single rows

COUNT, SUM, AVG, VARIANCE, MIN, MAX are standard functions sometimes also statistical functions (e.g. variance)

Using aggregation

```
SELECT to_Char( AVG (population), '9999999999999.99' ),  
       to_char(VARIANCE(population), '999999999.99' ),  
       MAX(population)  
FROM country
```

Target list: **do not mix aggregation and attribute values:**

```
SELECT name, MAX(population)  
FROM country
```

Syntax error:

```
00937. 00000 - "not a single-group group function"
```

Extended SELECT list

Aggregate functions allow for SELECT-Blocks in target list
- provided *one* result value is guaranteed ("scalar expression").

"Total population of provinces per country":

```
SELECT code, name , (SELECT sum(population) AS EWZ
                      FROM Province r
                      WHERE r.country=c.code)
FROM Country c
```

Correlated - not a surprise....

Aggregation

How many provinces has China: easy

```
SELECT COUNT(*)  
  AS "Number of Provinces"  
FROM Province r  
  
WHERE c.code = 'D'
```

Table with number of provinces per country: no way

?

...	
IND	32
IL	5
CL	1
D	16
...	

Grouping of a table

...	
USA	MI
USA	MN
USA	MO
Venda	V
Vanuatu	VAN
Vietnam	VN
Volksrepublik_China	SHA
Volksrepublik_China	SHG
Volksrepublik_China	SHX
...	

... **GROUP BY** country.name

Grouping

Def.: Grouping partitions a table into groups (or subtables) in such a way, that each group has equal values column wise in all **grouping attributes**
GROUP BY <attr1>, ... <attrn>

The **result list** of a grouped table **may only contain grouping attributes or aggregations** over other attribute of the groups.

Grouping

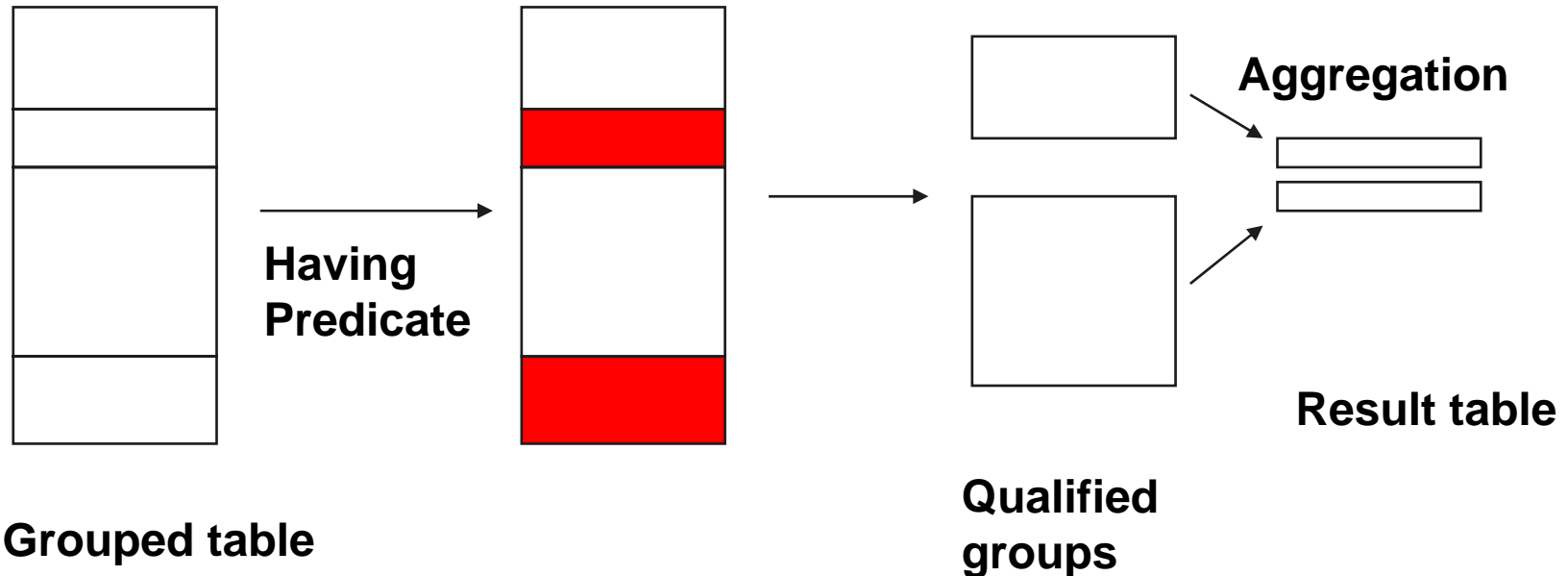
```
SELECT c.name, c.code, COUNT(*) AS noProvinces
FROM Country c JOIN Province r
      ON c.code=r.country
GROUP BY c.name, c.code
```

```
USA          USA    51
Vanuatu      VAN     1
Venda        V       1
Venezuela    YV      1
Vietnam      VN      1
Volksrepublik_China  VRC    27
```

- How to find the country with
- the maximum number of Provinces ?
 - or those with more than one Province?

Having

Group based selection:



```
SELECT c.name, c.code, COUNT(*) AS noRegs  
FROM Country c JOIN Province r ON c.code=r.country  
GROUP BY c.name, c.code  
HAVING COUNT(*) > 1
```

Having

**Predicates in having clause:
defined only on grouped columns or aggregated by
a set function**

```
SELECT c.name, c.code, SUM (ci.population) AS  
"Stadtbevoelkerung"  
FROM Country c JOIN City ci ON c.province =  
ci.province AND c.code=ci.country  
GROUP BY c.name, c.code  
HAVING SUM(ci.population) > 0
```

- "No HAVING without 'GROUP BY' "
- Attributes in Target list must be group attributes

Row and table predicates

Row predicates: evaluated for each individual row

Table predicates: evaluated on tables or groups.

" A group / table is qualified or not"

No aggregation in row predicates: MAX, COUNT() etc do not make any sense.

Aggregation mandatory for table predicates:

COUNT(*) > 2, MAX(population)

Brain teaser: can table predicates be expressed by row predicates?

Using group by

For readability, we introduce a **VIEW**:

```
CREATE VIEW NoReg AS (  
  SELECT c.name, c.code, COUNT(*) AS nofRegs  
  FROM Country c JOIN Province r ON c.code=r.country  
  GROUP BY c.name, c.code )
```

Standard step: **construct a joined table with all the information needed**. Join previous expression with **NoReg**

```
SELECT c.name, c.code, c.capital, c.province, ci.population  
FROM Country c JOIN City ci ON ci.name=c.capital  
  AND c.province = ci.province AND c.code=ci.country  
  JOIN NoReg n on n.name = c.name AND n.code = c.code  
WHERE n.nofRegs > 1      -- now a row predicate!  
AND ...- - WHERE EXISTS..
```

Expl. cont.

```
SELECT c.name, c.code, c.capital, c.province, ci.population
FROM Country c JOIN City ci ON ci.name=c.capital
    AND c.code = ci.province AND c.code=ci.country
    JOIN NoReg n on n.name = c.name AND n.C_ID = c.C_ID
WHERE n.nofRegs > 1      -- now a row predicate!
AND NOT EXISTS
    (SELECT *
    FROM Province r
    WHERE r.country = c.code AND c.province != r.province
    AND r.population > ci.population )
ORDER by c.name
```

NAME	C_ID	CAPITAL	R_ID	POPULATION
Daenemark	DK	Kopenhagen	DK	1358540
Grossbritannien	GB	London	ENG	6754500
...				
Völkrsrepublik_China	VRC	Peking	PEK	9900000

11 rows selected

Quantifiers and **counting** (in finite sets)

```
select x
from R
where EXISTS
    (select * from S...)
```

≡

```
select x
from R
where 0 <
    (select count(*) from S...)
```

```
SELECT DISTINCT country
FROM IsMember m1
WHERE 0 = (
    SELECT Count(*) FROM IsMember m2
    WHERE country = 'D' AND NOT EXISTS (
        SELECT * FROM IsMember m3
        WHERE m3.country = m1.country AND
            m3.organization = m2.organization
    )
)
```


GROUP BY

A realistic example¹

```
product (product_id, name, price, cost)
sales (product_id, units, date, ...)
```

"Find for each product the profit made within the last 4 weeks
if less than 500 \$ "

```
SELECT p.product_id, p.name,
       (sum(s.units) * (p.price - p.cost)) AS profit
FROM products p LEFT JOIN sales s USING (product_id)
WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY product_id, p.name, p.price, p.cost
HAVING sum(p.price * s.units) < 500;
```

¹ from the Postgres manual

SQL / DML Structuring

Temporary tables

When **inconsistency** is not an issue, temporary tables make sense

No assignment in SQL – applicative language

Instead: **Declare temporary relation** :

```
create temporary table myTmp ( .... )
```

and **assign a value** by means of an **INSERT** statement:

useful

variant of

insert

```
⇒ insert into mytmp  
   (select x,y,z from R where...)
```

Temporary tables are **local snapshots**, they are "dropped" at the end of a session.

Structuring

Consistency thread in multiuser mode!

Structuring

Subquery factoring / local definition

```
WITH r AS (  
    select  m.title, m.m_id AS x, tt.m_id  
    from movie m, DVD tt  
    where m.year > to_date(2000,'YYYY') )
```

```
SELECT DISTINCT r.title, t.DVD_id  
from r, DVD t  
where r.x = t.m_id;
```

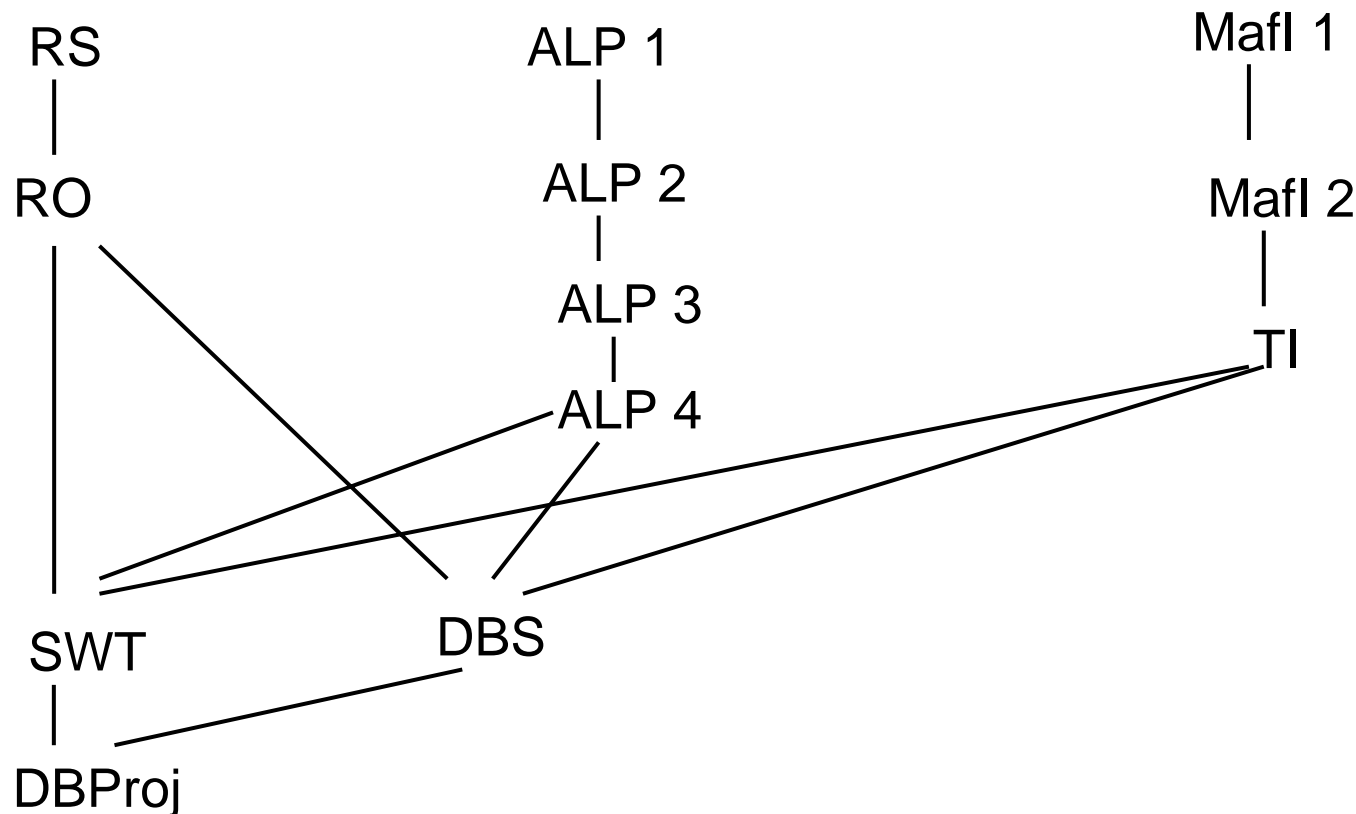
Local definition



Compare **let** in Haskell

Transitive closure

Representing a directed Graph
Example: **course prerequisites**



Represent graph by a set of nodes and a set of edges

Transitive closure

Example: Find courses required for SWT

enhanced
SQL:1999

```
-- Nodes
CREATE TABLE Course(
  lnr int primary key,
  name varchar(20));
```

```
-- Edges
CREATE TABLE Requires(
  pre int references course(lnr),
  suc int references course(lnr),
  constraint req_pk primary key(pre, suc));
```

ANSI SQL: Transitive closure

ANSI SQL (SQL 99) syntax for recursive traversals

```
WITH RECURSIVE PreCourse( pre, suc )
  AS (SELECT pre,suc FROM Requires r WHERE pre
      NOT IN (SELECT suc FROM Requires r1)
  UNION
      SELECT pre,suc
      FROM Requires r, PreCourses p
      WHERE p.suc = r.pre
  )
SELECT p1.suc , c.name
FROM preCourse p1, course c
WHERE      p1.suc = c.lnr;
```

Recursive processing

Querying a table recursively

(1) Construct the table which is recursively defined
Example: **PreCourses** (**pre**, **suc**) which is
the transitive closure of the **Requires** table

(1a) Start with the "base" relation **Requires**

Requires \rightarrow **PreCourses**⁰

(1b) construct **PreCourses**ⁿ⁺¹ :

PreCoursesⁿ union [all]

additional transitive dependent tuples using

Requires and **PreCourses**ⁿ

(2) Use constructed table (**PreCourses**) for querying

Termination?

The (iterative!) algorithm which constructs the transitive closure, terminates, if there are no new tuples to be added:

$$\text{PreCourses}^n = \text{PreCourses}^{n+1}$$

Crucial: The **result set** of the query defining the "delta" must **eventually be empty!**

In the example:

```
SELECT pre,suc
      FROM Requires r, PreCourses p
 WHERE p.suc = r.pre
```


Oracle: different mechanism

Transitive Closure in Oracle: CONNECT

Courses

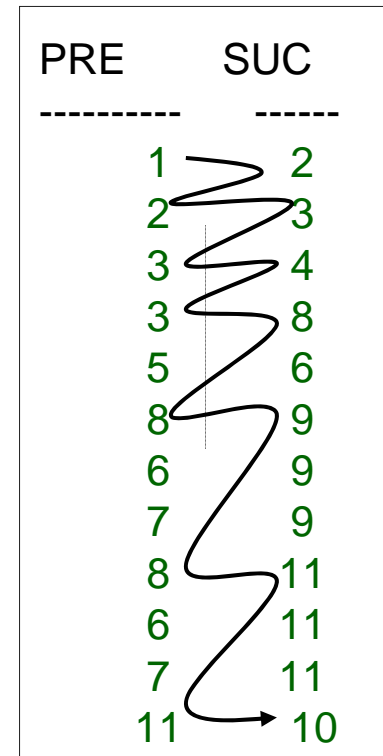
LNR NAME

```
1 ALP 1
2 ALP 2
3 ALP 3
4 ALP 4
5 RS
6 RO
7 Theory
8 SWP
9 DBS
10 DBProj
11 SWT
```

```
SELECT l.name
FROM Course l, requires r
WHERE r.suc = l.lnr
START WITH pre = 1
CONNECT BY PRIOR suc = pre;
```

NAME

```
-----
ALP 2
ALP 3
ALP 4
SWP
DBS
SWT
DBProj
```



1. Only ONE path ("start with..")
2. from leaf to root: exchange **suc** and **pre**

SQL / DML: Overview

Query data

- Interactively
- Embedded in host language
important in applications, next chapter

Insert, update, delete data

7.3 SQL / DML: Update operations



Delete, Insert, Update

The easiest way to ruin your company:

```
DELETE FROM Customer;
```

deletes all rows from **Customer** relation

In general, the rows to be deleted are specified by a (search) predicate:

```
DELETE FROM <tablename> WHERE <predicate>;
```

Very different from **deleting metadata**:

```
DROP TABLE Customers;    DROP SCHEMA my_database;
```

SQL / DML: Update

General form (simplified) for changing values :

```
UPDATE <tableName> SET
    <attr> = <value> [, <attr> = <value> ]0..*
    WHERE <searchPredicate>
```

- Note: without a predicate all rows will be changed
- **Primary key predicate for update** is very common

```
Update Customer SET email = 'me@acm.org'
    WHERE mem_No = 47.1;      /* primary key */
```

```
Update Rental SET until_date = SYSDATE
    WHERE bike_id = 23-7789 AND c_id = 3315
    AND until_date IS NULL;
```

SQL / DML: Insertion

```
INSERT INTO <tableName> VALUES  
[<value> [,value]] -- for each attribute
```

```
INSERT INTO Customer  
VALUES (011, 'Müller', 'Tina', NULL, NULL, NULL);
```

Insert (2)

Incomplete form with attribute and value list:

- Order of attributes / values independent from schema
- Attributes not in value list get value NULL

```
INSERT INTO <tableName>  
  (attribute [,attribute]0..n) VALUES  
  (<value> [,value]0..n)
```

```
INSERT INTO Customer  
(name, mem_no) VALUES ('Müller', 012);
```

SQL / DML: Insert data

Insertion using a query

```
INSERT INTO Foo (select * FROM Tmp)
```

Result set of query must have same type signature as table inserted to.

Bulk insertion

large file of INSERT statements may be inefficient

insertion of large data sets by specific DB tools

Postgres: COPY command to and from files (e.g. cvs)

Oracle and others: bulk loader

not standardized

SQL / DML: bulk load

Bulk load: inserting many data

Example:

```
CREATE TABLE loadtest(  
    name varchar(20),  
    num number(10,2));
```

loadtest.dat

```
'XYZ' , 4  
'YZX' , 5  
'ZXY' , 6
```

Oracle Syntax:

```
sqlldr <user>/<password> <controlfile>  
      <logfile> <badfile> <datafile>
```

loadtest.ctl

```
load data  
infile 'loadtest.dat'  
badfile 'loadtest.bad'  
discardfile 'loadtest.dis'  
APPEND INTO table loadtest  
fields terminated by " , "  
optionally enclosed by " ' "  
(name char, num integer external)
```


Summary

- SQL: **THE interlingua** of data management
- Differences (standard, systems) considerable
- Eventually convergence towards SQL 3
- Set manipulation as dominating operation
- Set specification in a declarative way
- Grouping: frequent operation
- Many language enhancements in SQL 3 (transitive closure, structuring)
- Interactive language: embedding into host language to be discussed