Introduction to Database Systems

Queries in SQL

Werner Nutt

The **`select`** Statement (Basic Version)

- Query statements in SQL start with the keyword

  `select`

  and return a result in table form

`select` *Attribute* … *Attribute*
`from` *Table* … *Table*
[**`where`** *Condition*]

- The three parts are usually called
  - **target list**
  - **from clause**
  - **where clause**

MotherChild

mother	child
Lisa	Mary
Lisa	Greg
Anne	Kim
Anne	Phil
Mary	Andy
Mary	Rob

FatherChild

father	child
Steve	Frank
Greg	Kim
Greg	Phil
Frank	Andy
Frank	Rob

Person

name	age	income
Andy	27	21
Rob	25	15
Mary	55	42
Anne	50	35
Phil	26	30
Greg	50	40
Frank	60	20
Kim	30	41
Mike	85	35
Lisa	75	87

Selection and Projection

Name and income of persons that are less than 30:

$$\pi_{\text{name, income}}(\sigma_{\text{age}<30}(\text{Person}))$$

```
select name, income
from   person
where  age < 30
```

name	income
Andy	21
Rob	15
Phil	30

Naming Conventions

- To avoid ambiguities, every attribute name has two components

  RelationName.AttributeName

- When there is no ambiguity, one can drop the initial component

  RelationName.

```
select person.name, person.income
from   person
where  person.age < 30

can be written as:

select name, income
from   person
where  age < 30
```

select: Abbreviations

```
select name, income
from   person
where  age < 30
```

is an abbreviation for:

```
select person.name, person.income
from   person
where  person.age < 30
```

and also for:

```
select p.name as name,  p.income as income
from   person p
where  p.age < 30
```

Two Kinds of Projection

Surname and branch of all employees

Employee

empNo	surname	branch	salary
7309	Black	York	55
5998	Black	Glasgow	64
9553	Brown	London	44
5698	Brown	London	64

$\pi_{\text{surname, branch}}$ (Employee)

Two Kinds of Projection

```
select
       surname, branch
from employee
```

surname	branch
Black	York
Black	Glasgow
Brown	London
Brown	London

```
select distinct
       surname, branch
from employee
```

surname	ranch
Black	York
Black	Glasgow
Brown	London

Usage of “as” in **`select`** Statements

“**`as`**” in the list of attributes specifies explicitly a name for the attributes of the result. If for some attribute “**`as`**” is missing, the name is equal to the one that appears in the list.

Example:

```
select name as personName, income as salary
from   person
where  age < 30
```

returns as result a relation with two attributes, the first having the name **`personName`** and the second having the name **`salary`**

```
select name, income
from   person
where  age < 30
```

returns as result a relation with two attributes, the first having the name **`name`** and the second having the name **`income`**

Exercise 1

“From the table **person,** compute a new table by selecting only the persons with an income between 20 and 30, and adding an attribute that has, for every tuple, the same value as **income**.

Show the result of the query”

Person

name	age	income

Exercise 1: Solution

```
select name, age, income,
       income as also-income
from   person
where  income >= 20 and income <= 30
```

name	age	income	also-income
Andy	27	21	21
Phil	26	30	30
Frank	60	20	20

Selection, without Projection

name, age and income of persons younger than 30:

$$\sigma_{age<30}(\text{Person})$$

```
select *
from   person
where  age < 30
```

all attributes

is an abbreviation for:

```
select name, age, income
from   person
where  age < 30
```

`select` with Asterisk

Given a relation **R** with attributes **A**, **B**, **C**

```
select  *
from    R
where   cond
```

is equivalent to

```
select A, B, C
from   R
where  cond
```

Projection without Selection

name and income of all persons

$$\pi_{\text{name, income}}(\text{Person})$$

```
select name, income
from   person
```

is an abbreviation for:

```
select p.name, p.income
from   person p
where  true
```

Expressions in the Target List

```
select income/4 as quarterlyIncome
from   person
where  name = ‘Greg'
```

Complex Conditions in the “**where**” Clause

```
select *
from   person
where  income > 25
       and (age < 30 or age > 60)
```

The “**like**” Condition

The persons having a name that starts with '**A**' and has a '**d**' as the third letter:

```
select *
from   person
where  name like 'A_d%‘
```

- ‘_‘ matches a single letter
- ‘**%**‘ matches a string

Handling of Null Values

Employees whose age is or could be greater than 40:

$$\sigma_{\text{age > 40 OR age IS NULL}} \text{ (Employee)}$$

```
select *
from   employee
where  age > 40 or age is null
```

Exercise 2

“From the table **`employee`**, calculate a new table by selecting only employees from the London and Glasgow branches, projecting the data on the attribute **`salary`**, and adding an attribute that has, for every tuple, twice the value of the attribute **`salary`**.

Show the result of the query”

Employee

empNo	surname	branch	salary

Exercise 2: Solution

```
select salary,
       salary*2 as doubleSalary
from   employee
where  branch = ‘Glasgow' or
       branch = ‘London'
```

salary	doubleSalary
64	128
44	88
64	128

Selection, Projection, and Join

- Using **`select`** statements with a single relation in the **`from`** clause we can realise:
  - selections,
  - projections,
  - renamings

- **Joins** (and Cartesian products) are realised by using two or more relations in the **`from`** clause

SQL and Relational Algebra

Given the relations: R1(A1,A2) and R2(A3,A4)

the semantics of the query

```
select R1.A1, R2.A4
from   R1, R2
where  R1.A2 = R2.A3
```

can be described in terms of

- Cartesian product (**`from`**)
- selection (**`where`**)
- projection (**`select`**)

Note: This does not mean that the system really calculates the Cartesian product!

SQL and Relational Algebra (cntd)

Given the relations: R1(A1,A2) and R2(A3,A4),

```
select R1.A1, R2.A4
from   R1, R2
where  R1.A2 = R2.A3
```

corresponds to :

$$\pi_{A1,A4} (\sigma_{A2=A3} (R1 \times R2))$$

SQL and Relational Algebra (cntd)

It may be necessary to rename attributes

- in the target list (as in relational algebra)
- in the Cartesian product (in particular, when the query refers twice to the same table)

```
select X.A1 as B1, ...
from   R1 X, R2 Y, R1 Z
where  X.A2 = Y.A3 and ...
```

which can also be written as

```
select X.A1 as B1, ...
from   R1 as X, R2 as Y, R1 as Z
where  X.A2 = Y.A3 and ...
```

SQL and Relational Algebra (cntd)

```
select X.A1 as B1, Y.A4 as B2
from   R1 X, R2 Y, R1 Z
where  X.A2 = Y.A3 and Y.A4 = Z.A1
```

$X \leftarrow R1, Y \leftarrow R2, Z \leftarrow R1,$

$$\rho_{B1 \leftarrow X.A1,\ B2 \leftarrow Y.A4} (\pi_{X.A1, Y.A4} (\sigma_{X.A2 = Y.A3 \text{ and } Y.A4 = Z.A1} (X \times Y \times Z)))$$

MotherChild

mother	child
Lisa	Mary
Lisa	Greg
Anne	Kim
Anne	Phil
Mary	Andy
Mary	Rob

FatherChild

father	child
Steve	Frank
Greg	Kim
Greg	Phil
Frank	Andy
Frank	Rob

Person

name	age	income
Andy	27	21
Rob	25	15
Mary	55	42
Anne	50	35
Phil	26	30
Greg	50	40
Frank	60	20
Kim	30	41
Mike	85	35
Lisa	75	87

Exercise 3

“The fathers of persons who earn more than 20K”

Write the query both in relational algebra and SQL

Exercise 3: Solution

"The fathers of persons who earn more than 20K"

$\pi_{father}(FatherChild \bowtie_{child=name} \sigma_{income>20}(Person))$

```
select distinct fc.father
from   person p, fatherChild fc
where  fc.child = p.name
       and p.income > 20
```

Exercise 4: Join

“Father and mother of every person”

Write the query both in relational algebra and SQL

Exercise 4: Solution

“Father and mother of every person”

Can be calculated in relational algebra by means of a natural join

$$\text{FatherChild} \bowtie \text{MotherChild}$$

```
select fc.child, fc.father, mc.mother
from   motherChild mc, fatherChild fc
where  fc.child = mc.child
```

Exercise 4: Join and Other Operations

“Persons that earn more than their father,
showing name, income, and income of the father”

Write the query both in relational algebra and SQL

Exercise 5: Solution

"Persons that earn more than their father,
showing name, income, and income of the father"

$$\pi_{\text{name, income, FI}} (\sigma_{\text{income>FI}} (\rho_{\text{FN} \leftarrow \text{name, FA} \leftarrow \text{age, FI} \leftarrow \text{income}}(\text{Person}) \bowtie_{\text{FN=father}} (\text{FatherChild} \bowtie_{\text{child =name}} \text{Person})))$$

```
select c.name, c.income, f.income
from   person f, fatherChild fc, person c
where  f.name = fc.father and
       fc.child = c.name and
       c.income > f.income
```

select, with Renaming of the Result

For the persons that earn more than their father, show their name, income, and the income of the father

```
select fc.child, c.income as income,
       f.income as incomefather
from   person f, fatherChild fc, person c
where  f.name = fc.father and
       fc.child = c.name and
       c.income > f.income
```

Explicit Join

For every person, return the person, their father and their mother

```
select fatherChild.child, father, mother
from   motherChild, fatherChild
where  fatherChild.child = motherChild.child
```

```
select fatherChild.child, father, mother
from   motherChild join fatherChild on
       fatherChild.child = motherChild.child
```

select with Explicit Join, Syntax

```
select …
from   Table { join Table on JoinCondition }, …
[ where OtherCondition ]
```

Exercise 6: Explicit Join

"For the persons that earn more than their father, show their name, income, and the income of the father"

Express the query in SQL, using an explicit join

Exercise 6: Solution

"For the persons that earn more than their father, show their name, income, and the income of the father"

```
select c.name, c.income, f.income
from   person c
          join fatherChild fc on c.name = fc.child
          join person f on fc.father = f.name
where  c.income > f.income
```

An equivalent formulation without explicit join:

```
select c.name, c.income, f.income
from   person c, fatherChild fc, person f
where  c.name = fc.child and
       fc.father = f.name and
       c.income > f.income
```

A Further Extension: Natural Join (Less Frequent)

"Return the names of fathers, mothers, and their children"

$$\pi_{\text{father,mother,child}}\,(\text{FatherChild} \bowtie \text{MotherChild})$$

In SQL:

```
select father, mother, fatherChild.child
from   motherChild join fatherChild on
              fatherChild.child = motherChild.child
```

Alternatively:

```
select father, mother, fatherChild.child
from   motherChild natural join fatherChild
```

Outer Join

“For every person, return the father and, if known, the mother”

```
select fatherChild.child, father, mother
from   fatherChild left outer join motherChild
       on fatherChild.child = motherChild.child
```

Note: “**`outer`**” is optional

```
select fatherChild.child, father, mother
from   fatherChild left join motherChild
       on fatherChild.child = motherChild.child
```

Outer Join: Examples

```
select fatherChild.child, father, mother
from   motherChild join fatherChild
       on motherChild.child = fatherChild.child

select fatherChild.child, father, mother
from   motherChild left outer join fatherChild
       on motherChild.child = fatherChild.child

select fatherChild.child, father, mother
from   motherChild right outer join fatherChild
       on motherChild.child = fatherChild.child

select fatherChild.child, father, mother
from   motherChild full outer join fatherChild
       on motherChild.child = fatherChild.child
```

Ordering the Result: **order by**

"Return name and income of persons under thirty, in alphabetic order of the names"

```
select name, income
from   person
where  age < 30
order by  name
```

ascending order

```
select name, income
from   person
where  age < 30
order by  name desc
```

descending order

Ordering the Result: **`order by`**

```
select name, income
from   person
where  age < 30
```

name	income
Andy	21
Rob	15
Mary	42

```
select name, income
from   person
where  age < 30
order by name
```

name	income
Andy	21
Mary	42
Rob	15

Aggregate Operators

Among the expressions in the target list, we can also have expressions that calculate values based on multisets of tuples:

– count, minimum, maximum, average, sum

Basic Syntax (simplified):

Function **(** [**`distinct`**] *ExpressionOnAttributes* **)**

Aggregate Operators: `count`

Syntax:

- counts the number of tuples:

  `count (*)`

- counts the values of an attribute (considering duplicates):

  `count (`*Attribute***`)`**

- counts the distinct values of an attribute:

  `count (distinct` *Attribute***`)`**

Aggregate Operator **count**: Example

***Example*:** How many children has Frank?

```
select count(*) as NumFranksChildren
from   fatherChild
where  father = 'Frank'
```

Semantics: The aggregate operator (**count**), which counts the tuples, is applied to the result of the query:

```
select *
from   fatherChild
where  father = 'Frank'
```

Results of **`count`**: Example

FatherChild

father	**child**
Steve	Frank
Greg	Kim
Greg	Phil
Frank	Andy
Frank	Rob

NumFranksChildren
2

`count` and Null Values

```
select count(*)
from   person
```

Result = number of tuples
= 4

```
select count(income)
from   person
```

Result = number of values different from NULL
= 3

```
select count(distinct income)
from   person
```

Result = number of distinct values (excluding NULL)
= 2

Person

name	age	income
Andy	27	21
Rob	25	NULL
Mary	55	21
Anne	50	35

Other Aggregate Operators

`sum`, **`avg`**, **`max`**, **`min`**

- argument can be an attribute or an expression (but not "**`*`**")
- **`sum`** and **`avg`**: of numeric types and time intervals
- **`max`** and **`min`**: on types for which an ordering is defined: numbers, strings, time intervals, arrays

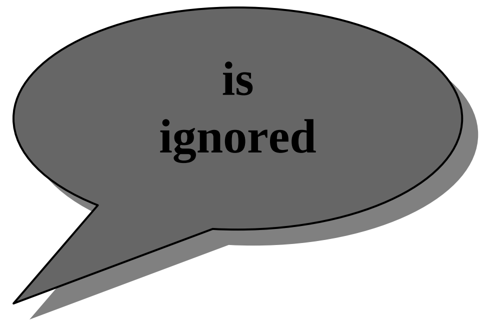
***Example*:** Average income of Frank's children

```
select avg(p.income)
from   person p join fatherChild fc on
       p.name = fc.child
where  fc.father = 'Frank'
```

Aggregate Operators and Null Values

```
select avg(income) as meanIncome
from   person
```

Person	name	age	income
	Andy	27	30
	Rob	25	NULL
	Mary	55	36
	Anne	50	36



meanIncome
34

Aggregate Operators and the Target List

An incorrect query (whose name should be returned?):

```
select name, max(income)
from   person
```

The **target list** has to be **homogeneous**, for example:

```
select min(age), avg(income)
from   person
```

Aggregate Operators and Grouping

- Aggregation functions can be applied to partitions of the tuples of a relations
- To specify the partition of tuples, one uses the **`group by`** clause:

`group by` *attributeList*

Aggregate Operators and Grouping

The number of children of every father.

```
select father, count(*) as NumChildren
from   fatherChild
group by father
```

FatherChild

father	child
Steve	Frank
Greg	Kim
Greg	Phil
Frank	Andy
Frank	Rob

father	NumChildren
Steve	1
Greg	2
Frank	2

Semantics of Queries with Aggregation and Grouping

1. The query is run **ignoring the `group by` clause** and the aggregate operators:

```
select *
from   fatherChild
```

2. The **tuples that have the same value for the attributes appearing in the `group by` clause**, are grouped into equivalence classes.

3. **Each group** contributes **a tuple** to the answer. The tuple consists of the **values of the `group by` attributes** and the result of **applying the aggregation function** to the group.

Exercise 7: `group by`

"For each group of adult persons who have the same age, return the maximum income for every group and show the age"

Write the query in SQL!

Person	name	age	income

Exercise 7: Solution

“For each group of adult persons who have the same age, return the maximum income for every group and show the age”

```
select age, max(income)
from   person
where  age > 17
group by age
```

Grouping and Target List

In a query that has a **`group by`** clause, **only** such attributes can appear in the target list (except for aggregation functions) the appear in the **`group by`** clause.

***Example*: Incorrect**: income of persons, grouped according to age

```
select age, income
from   person
group by age
```

There could exist several values for the same group.

Correct: average income of persons, grouped by age.

```
select age, avg(income)
from   person
group by age
```

Grouping and Target List (cntd)

The syntactic restriction on the attributes in the select clause holds also for queries that would be semantically correct (i.e., for which there is only a single value of the attribute for every group).

***Example*:** Fathers with their income and with the average income of their children.

Incorrect:

```
select fc.father, avg(c.income), f.income
from   person c join fatherChild fc on c.name=fc.child
                join person f on fc.father=f.name
group by fc.father
```

Correct:

```
select fc.ather, avg(c.income), f.income
from   person c join fatherChild fc on c.name=fc.child
                join person f on fc.father=f.name
group by fc.father, f.income
```

Conditions on Groups

It is also possible to **filter the groups** using selection conditions. Clearly, the selection of groups differs from the selection of the tuples in the **`where`** clause: the tuples form the groups.

To filter the groups, the “having clause” is used.

The having clause must appear after the “**`group by`**”

Example**:** “Fathers whose children have an average income greater 25.”

```
select fc.father, avg(c.income)
from   person c join fatherChild fc
       on c.name = fc.child
group by fc.father
having avg(c.income) > 25
```

Exercise 8: `where` or `having`?

“Fathers whose children under age 30 have an average income greater 20”

Exercise 8: Solution

“Fathers whose children under the age of 30 have an average income greater 20”

```
select father, avg(f.income)
from   person c join fatherChild fc
       on c.name = fc.child
where  c.age < 30
group by cf.father
having avg(c.income) > 20
```

Syntax of SQL `select` (Summary)

SQLSelect ::=

```
select      ListOfAttributesOrExpressions
from        ListOfTables
[ where     ConditionsOnTuples ]
[ group by  ListOfGroupingAttributes ]
[ having    ConditionsOnAggregates ]
[ order by  ListOfOrderingAttributes ]
```

Union, Intersection, and Difference

Within a **`select`** statement one cannot express unions.

An explicit construct is needed:

```
select ...
union [all]
select ...
```

With **`union`**, duplicates are eliminated (also those originating from projection).

With **`union all`** duplicates are kept.

Positional Notation of Attributes

```
select father, child
from   fatherChild
union
select mother, child
from   motherChild
```

Which are the attribute names of the result?

Those of the first operand!

→ SQL matches attributes in the same position

→ SQL renames the attributes of the second operand

Result of the Union

father	child
Greg	Frank
Greg	Kim
Greg	Phil
Frank	Andy
Frank	Rob
Lisa	Mary
Lisa	Greg
Anne	Kim
Anne	Phil
Mary	Andy
Mary	Rob

Positional Notation: Example

```
select father, child
from   fatherChild
union
select mother, child
from   motherChild
```

```
select father, child
from   fatherChild
union
select child, mother
from   motherChild
```

Positional Notation (cntd)

Renaming does not change anything:

```
select father as parent, child
from   fatherChild
union
select child, mother as parent
from   motherChild
```

Correct (if we want to treat fathers and mothers as parents):

```
select father as parent, child
from   fatherChild
union
select mother as parent, child
from   motherChild
```

Difference

```
select name
from   employee
except
select lastName as name
from   employee
```

We will see that differences can also be expressed with nested **`select`** statements.

Intersection

```
select name
from   employee
intersect
select lastName as name
from   employee
```

is equivalent to

```
select en.name
from   employee en, employee eln
where  en.name = eln.lastName
```

Single Block Queries: Exercises

Consider a database about suppliers and parts with the following schema:

```
Supplier(sid, sname, address)
Part(pid, pname, colour)
Catalog(sid, pid, cost)
```

Formulate the following queries in SQL:

1. Find the names of suppliers who supply some red part.

2. Find the IDs of suppliers who supply some red or green part.

3. Find the IDs of suppliers who supply some red part and are based at 21 George Street.

Single Block Queries: Exercises (cntd)

4. Find the names of suppliers who supply some red part or are based at 21 George Street.

5. Find the IDs of suppliers who supply some red and some green part.

6. Find pairs of IDs such that for some part the supplier with the first ID charges more than the supplier with the second ID.

7. For each supplier, return the maximal and the average cost of the parts they offer.

8. List those red parts that on average cost no more than 30 Euro.

9. List the names of those red parts that are offered by at least three suppliers.

Nested Queries

- In the atomic conditions of the **`where`** clause one can also use a **`select`** clause (which must appear in parentheses).

- In particular, in atomic conditions one can have:
  - comparisons of an attribute (or several attributes) with the result of a subquery
  - existential quantification

Nested Queries (Example)

“Name and income of Frank’s father”

```
select  f.name, f.income
from    person f, fatherChild fc
where   f.name = fc.father and fc.child = 'Frank'
```

```
select  f.name, f.income
from    person f
where   f.name = (select fc.father
                  from   fatherChild fc
                  where  fc.child = 'Frank')
```

Nested Queries: Operators

In the **where** clause, the result of a nested query can be related to other values by way of several **operators**:

- equality and other comparisons
  (the result of the nested query must be unique)
- if it is not certain that the result of the nested query is unique, the nested query can be preceded by one of the keywords:
  - **any**: true, if the comparison is true for **at least one** of the result tuples of the nested query
  - **all**: true, if the comparison is true for **all** the result tuples of the nested query
- the operator **in**, which is equivalent to **=any**
- the operator **not in**, which is equivalent to **<>all**
- the operator **exists**

Nested Queries: Example

Name and income of the fathers of persons who earn more than 20k.

```
select distinct f.name, f.income
from   person f, fatherChild fc, person c
where  f.name = fc.father and
       fc.child = c.name and c.income > 20
```

```
select f.name, f.income
from   person f
where  f.name = any
           (select fc.father
            from   fatherChild fc, person c
            where  fc.child = c.name and
                   c.income > 20)
```

fathers of persons who earn more than 20k

Nested Queries: Example

Name and income of the fathers of persons who earn more than 20k.

```
select f.name, f.income
from   person f
where  f.name in (select fc.father
                  from   fatherChild fc, person c
                  where  f[illegible]name
                         [illegible]20)
```

fathers of persons who earn more than 20k

```
select f.name, f.income
from   person f
where  f.name in (select fc.father
                  from  fatherChild fc
                  where fc child in (select c.name
                                     from   person c
                                     where c.income > 20)
                  )
```

persons who earn more than 20k

Nested Queries: Comments

- The nested formulation of a query is sometimes executed less efficiently than an equivalent unnested formulation (due to limitations of the query optimizer).

- The nested formulation is sometimes more readable.

Nested Queries: Example with **`all`**

“Persons who have an income that is higher than the income of all persons younger than 30”

Nested Queries: Example with **`all`**

“Persons who have an income that is higher than the income of all persons younger than 30”

```
select name
from   person
where  income >= all (select income
                      from   person
                      where  age < 30)
```

Equivalent Formulation with **`max`**

“Persons who have an income that is higher than the income of all persons younger than 30”

```
select name
from   person
where  income >= (select max(income)
                  from   person
                  where  age < 30)
```

Nested Queries: Example with **`exists`**

An expression with the operator **`exists`** is true if the result of the subquery is **not empty**.

Example**:** "Persons with at least one child"

```
select *
from   person p
where  exists (select *
               from  fatherChild fc
               where fc.father = p.name)
       or
       exists (select *
               from  motherChild mc
               where mc.mother = p.name)
```

Note: the attribute **`name`** refers to the table in the outer **`from`** clause.

Nesting, Union, and “or”

The query for “persons with at least one child” can also be expressed as a union:

```
select p.name, p.age, p.income
from   person p, fatherChild fc
where  fc.father = p.name
union
select p.name, p.age, p.income
from   person p, motherChild mc
where  mc.mother = p.name
```

Does the following query with “or” return the same answers?

```
select distinct p.name, p.age, p.income
from   person p, fatherChild fc, motherChild mc
where  fc.father = p.name
         or mc.mother = p.name
```

Nested Queries and Negation

All the queries with nesting in the previous examples are equivalent to some unnested query. So, what's the point of nesting?

Example**:** "Persons without a child"

```
select *
from   person p
where  not exists (select *
                   from  fatherChild fc
                   where fc.father = p.name)
       and
       not exists (select *
                   from  motherChild mc
                   where mc.mother = p.name)
```

This cannot be expressed equivalently as a "**`select from where`**" query.

Why?

Exercise 9

"Name and age of the mothers all of whose children are at least 18"

Approach 1: Subquery with **`all`**

Approach 2: Subquery with **`min`**

Approach 3: Subquery with **`not exists`**

Exercise 9: Solution with **`all`**

“Name and age of the mothers all of whose children are at least 18”

```
select m.name, m.age
from   person m join motherChild mc
                on m.name = mc.mother
where  18 =< all (select c0.age
                  from   motherChild mc0 join person c0
                               on mc0.mother = c0.name
                  where  mc0.mother = mc.mother)
```

Exercise 9: Solution with **min**

"Name and age of the mothers all of whose children are at least 18"

```
select m.name, m.age
from   person m join motherChild mc
                on m.name = mc.mother
where  18 =< (select min(c0.age)
              from   motherChild mc0 join person c0
                           on mc0.mother = c0.name
              where  mc0.mother = mc.mother)
```

"Name and age of mothers where the minimal age of their children is greater or equal 18"

Exercise 9: Solution with **not exists**

“Name and age of the mothers all of whose children are at least 18”

```
select m.name, m.age
from   person m join motherChild mc
                on m.name = mc.mother
where  not exists
             (select *
              from   motherChild mc0 join person c0
                           on mc0.mother = c0.name
              where  mc0.mother = mc.mother and
                     c0.age < 18)
```

Name and age of mothers who don’t have a child that is younger than 18.

Nested Queries: Comments

- **Visibility** rules:
  - it is not possible to refer to a variable defined in a block below the current block
  - if an attribute name is not qualified with a variable or table name, it is assumed that it refers to the “closest” variable or table with that attribute

- In each block, one can refer to variables defined in the same block or in surrounding blocks

- **Semantics**: the inner query is executed for every tuple of the outer query

Exercise

On the supplier and parts DB:

```
Supplier(sid, sname, address)
Part(pid, pname, colour)
Catalog(sid, pid, cost)
```

1. Suppliers that supply *only* red parts

2. Suppliers that supply *all* red parts

Nested Queries: Visibility

Persons having at least one child.

```
select *
from   person
where  exists (select *
               from  fatherChild
               where father = name)
       or
       exists (select *
               from  motherChild
               where mother = name)
```

The attribute **name** refers to the table **person** in the outer **from** clause.

More on Visibility

Note: This query is incorrect:

```
select *
from  employee
where dept in (select name
               from  department D1
               where name = 'Production')
      or
      dept in (select name
               from  department D2
               where D2.city = D1.city)
```

employee	name	lastName	dept

department	name	address	city

Visibility: Variables in Internal Blocks

Name and income of the fathers of persons who earn more than 20k, **showing also the income of the child**.

```
select distinct f.name, f.income, c.income
from   person f, fatherChild, person c
where  f.name = fc.father and fc.child = c.name
       and c.income > 20
```

In this case, the “intuitive” nested query is incorrect:

```
select name, income, c.income
from   person
where  name in (select father
                from fatherChild
                where child in (select name
                                from person c
                                where c.income > 20))
```

Correlated Subqueries

It may be necessary to **use in inner blocks** variables that are **defined in outer blocks**. In this case one talks about **correlated** subqueries.

Example: The fathers all of whose children earn strictly more than 20k.

```
select distinct fc.father
from   fatherChild fc
where  not exists (select *
                   from  fatherChild fc0, person c
                   where fc.father = fc0.father
                         and fc0.child = c.name
                         and c.income <= 20)
```

Exercise 10: Correlated Subqueries

“Name and age of mothers who have a child whose age differs less than 20 years from their own age”

Exercise 10: Solution

"Name and age of mothers who have a child whose age differs less than 20 years from their own age"

```
select m.name, m.age
from   person m, motherChild mc
where  m.name = mc.mother and
       mc.child in (select c.name
                    from   person c
                    where  m.age – c.age < 20)
```

Question: Intersection

Can one express intersection by way of nesting?

```
select name from employee
  intersection
select lastName as name from employee
```

Intersection by Way of Nesting

```
select name from employee
  intersection
select lastName as name from employee


select name
from  employee
where name in (select lastName
               from   employee)

select name
from  employee e
where exists (select *
              from   employee
              where  lastName = e.name)
```

Intersection Without Nesting

Is it possible to express intersection without nesting?

```
select name from employee
  intersection
select lastName as name from employee
```

Exercise 11

Can one express set difference by way of nesting?

```
select name from employee
  except
select lastName as name from employee
```

Exercise 11 (Solution 1)

Can one express set difference by way of nesting?

```
select name from employee
  except
select lastName as name from employee
```

```
select name
from  employee
where name not in (select lastName
                   from employee)
```

Exercise 11 (Solution 2)

Can one express set difference by way of nesting?

```
select name from employee
  except
select lastName as name from employee



select name
from   employee e
where  not exists (select *
                   from  employee
                   where lastName = e.name)
```

Exercise 12: Nesting and Functions

"The person (or the persons) that have the highest income"

Exercise 12: Solution

"The person (or the persons) that have the highest income"

```
select *
from  person
where income = (select max(income)
                from   person)
```

Or:

```
select *
from  person
where income >= all (select income
                     from   person)
```

Nested Queries: Conditions on Several Attributes

The persons which have a unique combination of age and income

(that is, persons for whom the pair (age, income) is different from the corresponding pairs of all other persons).

```
select *
from  person p
where (age,income) not in
                      (select age, income
                       from person
                       where name <> p.name)
```

Views

- A view is a table **whose instance is derived from other tables by a query.**

  create view *ViewName* [**(***AttributeList***)**] **as** *SQLSelect*

- Views are virtual tables: their instances (or parts of them) are only calculated when they are used (for instance in other queries).

Example:

```
create view AdminEmp(empNo,firstName,lastName,sal) as
  select EmpNo, firstName, lastName, salary
  from   employee
  where  dept = 'Administration' and
         salary > 10
```

Maximizing Aggregates

- "Which age group has the highest total income?"

- One solution is to use nesting in the **having** clause:

```
select age
from   person
group by age
having sum(income) >= all (select sum(income)
                           from   person
                           group by age)
```

- Another solution is to create a view.

Solution with Views

```
create view ageIncome(age,sumIncome) as
   select age, sum(income)
   from   person
   group by age


select age
from   ageIncome
where  sumIncome = (select max(sumIncome)
                    from   ageIncome)
```

Exercise 13

- Among all companies based in George Street that sell red parts, which is the one with the least average price for red parts?

Exercise 13 (Solution)

- Among all companies based in George Street that supply red parts, which is the one with the least average price for red parts?

```
create view RedPartCompGS(sid,name,avgCost) as
 select sid, name, avg(cost)
 from   supplier natural join catalog
                 natural join part
 where  address LIKE ‘%George St%’ AND
        colour = ‘red’
 group by sid, name
```

Exercise 13 (Solution, cntd)

- Among all companies based in George Street that sell red parts, which is the one with the least average price for red parts?

```
select name
from   RedPartCompGS
where  avgCost = (select max(avgCost)
                  from   RedPartCompGS)
```

Views Can Replace Subqueries

```
select *
from   person
where  name in (select father from fatherChild);
```

With a view

```
create view father(name) as
select distinct father from fatherChild;
```

```
select *
from   person
where  name in (select name from father);
```

Inline Views: Views in the **FROM** Clause

An equivalent formulation

```
select person.*
from   person, father
where  person.name = father.name;
```

If we need a view only once, we can define it in the FROM clause

```
select *
from   person,
       (select distinct father as name
        from   fatherChild) father
where  person.name = father.name;
```

Inline Views (Cntd)

Inline views can also take part in joins

```
select person.*
from   person
       natural join
       (select distinct father as name
        from   fatherChild) father;
```

Note: The inline view needs to be named,
even if the name is never used.

Inline Views in Aggregation of Aggregates

"The age group with the highest average salary"

```
select  age, avginc
from   (select   age, avg(income) as avginc
        from     person
        group by age) ageavg
where   ageavg.avginc = (select max(avginc)
                         from   (select avg(income)
                                        as  avginc
                                 from   person
                                 group by age)
                                             ageavgs);
```

This is not best practice! …but illustrates what is possible

Exercise

Consider the table

```
Employee(enumber, name, job, hiredate,
         salary, depno)
```

Task: Compute a new table giving data about employees per department for the job types of analysts and clerks
with the schema

```
(depno, n_clerks, n_analysts, tot_ca)
```

Step 1: Count Employees per Department and Job

```
select   depno, job, count(*)
from     employee
group by depno, job
order by depno, job;
```

Create a view that does this

```
create view depjobcount(depno, job, jobcount) as
select   depno, job, count(*)
from     employee
group by depno, job;
```

Step 2, Attempt 1: Join counts of Analysts and Clerks

```
select adjc.depno,
              adjc.jobcount as N_analysts,
              cdjc.jobcount as N_clerks
from depjobcount adjc
     join depjobcount cdjc on adjc.depno = cdjc.depno
where adjc.job = 'Analyst' and
      cdjc.job = 'Clerk';
```

What if there is a department without analyst or clerk?

Step 2, Attempt 2: Take the Outer Join

```
select adjc.depno,
              adjc.jobcount as N_analysts,
              cdjc.jobcount as n_Clerks
from depjobcount adjc
     full outer join depjobcount cdjc
                on adjc.depno = cdjc.depno
where adjc.job = 'Analyst' and
      cdjc.job = 'Clerk';
```

Is the result different this time?

Step 2, Attempt 3: Take the Outer Join of Inline Views

```
select adjc.depno,
       adjc.jobcount as N_analysts,
       cdjc.jobcount as N_clerks
from (select * from depjobcount
               where job = 'Analyst') adjc
     on alldeps.depno = adjc.depno
     full outer join
     (select * from depjobcount
               where job = 'Clerk') cdjc
     on alldeps.depno = cdjc.depno;
```

What about the department numbers?

Step 2, Attempt 4: Include all Departments

```
select alldeps.depno,
       adjc.jobcount as N_analysts,
       cdjc.jobcount as N_clerks
from (select distinct depno from depjobcount) alldeps
     full outer join
     (select * from depjobcount
               where job = 'Analyst') adjc
     on alldeps.depno = adjc.depno
     full outer join
     (select * from depjobcount
               where job = 'Clerk') cdjc
     on alldeps.depno = cdjc.depno;
```

Step 2, Attempt 5: Turn Nulls Into 0s

```
select alldeps.depno,
       coalesce(adjc.jobcount,0) as N_Analysts,
       coalesce(cdjc.jobcount,0) as N_Clerks
from (select distinct depno from depjobcount) alldeps
     full outer join
     (select * from  depjobcount
               where job = 'Analyst') adjc
     on alldeps.depno = adjc.depno
     full outer join
     (select * from depjobcount
               where job = 'Clerk') cdjc
     on alldeps.depno = cdjc.depno;
```

Step 3: Join With Total Number of Analysts and Clerks

Exercise …

Generic Integrity Constraints: **check**

Constraints on tuples or complex constraints on a single table are specified as:

check (*Condition***)**

```
create table Employee
( EmpNo     character(6),
  FirstName character(20),
  LastName  character(20),
  Sex       character not null check (sex in (‘M’,‘F’))
  Salary    integer,
  Superior  character(6),
  check (salary <= (select s.salary
                    from   employee s
                    where  superior = s.EmpNo))
)
```

When do you expect that this constraint will be checked?
Would that correspond to the semantics you expect?

Check Constraints (cntd)

- In systems, only check constraints are completely supported that need information from a single tuple, for instance

```
create table Employee
( EmpNo     character(6),
  FirstName character(20),
  LastName  character(20),
  Sex       character not null check (sex in ('M','F'))
  Salary    integer,
  Superior  character(6),
  check (not LastName = 'Smith' or Salary >= 40)
)
```

Generic Integrity Constraints: Assertions

Specify constraints at schema level. Syntax:

create assertion *AssName* **check (** *Condition* **)**

Example:

```
create assertion AtleastOneEmployee
check (1 <= (select count(*)
             from   employee))
```

– No efficient implementation techniques exist to date
– Systems do not support assertions

Access Control

- In SQL it is possible to specify
  - who can use (i.e., which user)
  - in which way (i.e., read, write,...)

  a data base (or part of it)

- The object of such **privileges** (access rights) are usually tables, but also other types of **resources**, like attributes, views, or domains.

- The predefined user **`_system`** (database administrator) has all privileges.

- The creator of a resource has all privileges for it.

Characteristics of Privileges

A privilege is characterised by:

- the resource to which it refers
- the user who grants the privilege
- the user who receives the privilege
- the action that is permitted
- the possibility to transfer the privilege

Privileges (cntd)

Types of privileges

- **insert**: permits to insert new records (tuples)
- **update**: permits to modify the content
- **delete**: permits to eliminate records
- **select**: permits to read the resource
- **references**: permits the definition of referential integrity constraints that target the resource (can limit the possibility to modify the resource)
- **usage**: permits the usage in a definition (for example, the usage of a domain)

grant and **revoke**

- **Concession** of privileges:

  grant < *Privileges* | **all privileges** > **on** *Resource* **to** *Users* [**with grant option**]

  - **with grant option** specifies whether the privilege can be transferred to other users

    grant select on Department to Joe

- **Revocation** of privileges:

  revoke *Privileges* **on** *Resource* **from** *Users* [**restrict** | **cascade**]

Transactions

A transaction is the execution of a program that accesses the DB and

- starts with a **`BEGIN`** operation
- followed by a number of SQL statements
- and ends with a **`COMMIT`** or **`ROLLBACK`** operation.

Example Transaction in SQL

```
begin transaction;
    update CurrentAccount
       set Balance = Balance – 10
       where AccountNo = 12345;
    update CurrentAccount
       set Balance = Balance + 10
       where AccountNo = 55555;
commit work;
```

What can go wrong during the execution of this transaction?

Transactions in SQL

Basic instructions:

- **`begin transaction`**: specifies the beginning of the transaction (the specified operations do not yet leave a permanent effect on the database itself, e.g., they are written into a log file)
- **`commit work`**: the operations specified after the **`begin transaction`** are being made permanent
- **`rollback work`**: the request to execute the operations after the last **`begin transaction`** is withdrawn

Savepoints

Savepoints allow one to limit rollbacks

```
begin transaction;
    update CurrentAccount
        set Balance = Balance – 10
        where AccountNo = 12345;
savepoint mysavepoint;
    update CurrentAccount
        set Balance = Balance + 10
        where AccountNo = 55555;
rollback to savepoint mysavepoint;
```

Transactions are Interleaved

- Large database systems are typically multi-user systems
  → *many transactions* are running at the *same time*

- Running transactions serially (i.e., one after the other) is inefficient:
  - transactions are often *waiting for I/O* to complete

  → serial execution leads to *low resource utilisation*

Transactions

- A transaction is a sequence of operations that is considered indivisible (“atomic”), that is not influenced during its execution by other operations on the database (“isolated”), and whose effects are definitive (“durable”).

- Properties (“**ACID**”):
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability (persistence)

We shall discuss these properties one by one

Transactions are … Atomic

- The sequence of operations on the database is either executed in its entirety, or not at all.

Example: transfer of funds from account A to account B: either both, the debit on A and the deposit into B are executed, or none of the two.

Transactions are … Consistent

- After a transaction has been executed, the integrity constraints have to be satisfied.
- During the execution, there may be violations, but if they remain until the end, the transaction has to be undone (“aborted”).

Transactions are … Isolated

- Transactions must not interfere with each other.
- The effect of a group of transactions on the database that are executed concurrently must be the same as the effect of some serial execution (i.e., as if they had been executed one after the other).

Example: A withdrawal from a bank account could interfere with a concurrent deposit so that the effect of one is overridden by the other.

Transactions are … Durable

- After the succesful completion of a transaction, the DBMS commits to make the outcome of the transaction permanent, even in the presence of concurrency and/or breakdowns

Exercise: ACID Properties of Transactions

Suppose a database system is used to organize the *check-in of airline passengers*:

- there is a *list of passengers*, and upon arrival a *seat* has to be assigned to each passenger and their *luggage* has to be checked in.

Briefly describe in this context the four ACID properties of transactions, i.e.,

- *for each property* give *an example* that illustrates the problem which such a system might suffer if the property is not supported.

Integrity Checking and Transactions

A company database has the following two tables

`Emp(empno int, ename string, depno int)`

`Dept(depno int, dname string, hod int)`

where **hod** stands for the "Head of Department",
who has to be an employee.

There are two referential integrity constraints

`Emp(depno) references Dept(depno)`

`Dept(hod) references Emp(empno)`

How can we define the two relations in SQL?

Integrity Checking and Transactions (cntd)

Table definition (Attempt 1)

```
CREATE TABLE Emp (
  empno int PRIMARY KEY,
  ename varchar(20),
  depno int,
  FOREIGN KEY (depno)
     REFERENCES Dept(depno));

CREATE TABLE Dept (
  depno int primary key,
  dname varchar(20),
  hod int,
  FOREIGN KEY (hod)
        REFERENCES Emp(empno));
```

What will the DBMS respond?

Integrity Checking and Transactions (cntd)

Table definition (Attempt 2)

```
CREATE TABLE Emp (
  empno int PRIMARY KEY,
  ename varchar(20),
  depno int);
```

```
CREATE TABLE Dept (
  depno int primary key,
  dname varchar(20),
  hod int);
```

```
ALTER TABLE Emp
  ADD CONSTRAINT emp_fk_dept
  FOREIGN KEY (depno)
       references Dept(depno);
```

```
ALTER TABLE Dept
  ADD CONSTRAINT dept_hod_is_emp
  FOREIGN KEY (hod)
       references Emp(empno);
```

Are both ALTER TABLE statements necessary?

Integrity Checking and Transactions (cntd)

Next, we want to populate the database:

Smith is the first employee,
Accounting the first department, and
Smith is the head of the accounting department:

```
INSERT INTO emp VALUES (1, 'Smith', 1);

INSERT INTO dept VALUES (1, 'Accounting', 1);
```

How will the DBMS react?

Integrity Checking and Transactions (cntd)

In SQL, integrity constraints can be declared as deferrable

(in PostgreSQL, this is only possible for foreign key constraints)

If a (deferrable) constraint is deferred during a transaction,
it is only checked at the end of the transaction.

We modify the constraint definitions:

```
ALTER TABLE Emp
  ADD CONSTRAINT emp_fk_dept
  FOREIGN KEY (depno)
       references Dept(depno)
  DEFERRABLE;
```

```
ALTER TABLE Dept
  ADD CONSTRAINT dept_hod_is_emp
  FOREIGN KEY (depno)
       references Dept(depno)
  DEFERRABLE;
```

Integrity Checking and Transactions (cntd)

Now, we can combine the insertion steps in one transaction:

```
BEGIN;
SET CONSTRAINTS emp_fk_dept DEFERRED;
SET CONSTRAINTS dept_hod_is_emp DEFERRED;
INSERT INTO emp VALUES (1, 'Smith', 1);
INSERT INTO dept VALUES (1, 'Accounting', 1);
COMMIT;
```

Was it necessary to defer both constraints?

Concurrency Control

- There are two owners of the new company, Alice and Bob, who are hiring staff.
- Alice wants to hire Black, Bob wants to hire Brown.
- The two are running concurrently two transaction on the company database

Alice	Bob
`begin;`	
	`begin;`
`insert(2,'Black',1)`	
	`insert(2,'Brown',1)`

How should the DBMS react?

What if Alice commits?

What if Alice does rollback, and then commits?

Concurrency Control (cntd)

- Alice has hired two new employees, McBlack and Mc Brown, while Bob has hired two other employees, OBlack and OBrown
- Alice and Bob are making the following insertions:

Alice	Bob
begin;	
	begin;
insert(2,'McBlack',1)	
	insert(3,'OBrown',1)
Insert(3,'McBrown',1)	
	insert(2,'OBlack',1)

What should the DMBS do?

Deadlocks

Deadlocks can be detected by maintaining a “Wait-For-Graph”:

- Waiting transactions are the nodes
- There is an edge from T_i to T_j if T_i is waiting for T_j

$\Rightarrow$ There is a deadlock among the waiting transactions
iff the Wait-For-Graph contains a cycle

To break the deadlock,
roll back enough transactions so that the cycle vanishes

In our example, the PostgreSQL server responds:

```
ERROR:  deadlock detected
DETAIL:  Process 15806 waits for ShareLock on transaction 16094;
blocked by process 15785.
Process 15785 waits for ShareLock on transaction 16095;
blocked by process 15806.
********** Error **********
```

Violations of Isolation

The SQL standard distinguishes between:

- Dirty read

  A transaction reads data written by a concurrent uncommitted transaction

- Nonrepeatable read

  A transaction re-reads a tuple it has previously read and finds that the tuple has been changed (modified or deleted) by another transaction (that committed since the initial read)

- Phantom read

  A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction

Isolation Levels Prevent Violations

Four isolation levels can be defined for a transaction

Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

- “Read committed“ is default
- Isolation levels are defined by

  `SET TRANSACTION ISOLATION LEVEL` *IsolationLevel*

Transactions (Example)

A sailing club allows its members to reserve individual boats over the internet for periods of several days. The club uses a DBMS and relies on the concurrency control of the DBMS to prevent conflicting reservations. The club‘s database contains a table with the schema

reservation(<u>boatName</u>, <u>startDate</u>, endDate, sailorName)

The table contains only the tuple

('Marine', '10-Apr-08', '13-Apr-08', 'Dustin')

Transactions (Example ctd)

Two sailors, Rusty and Lubber, connect to the database at the same time to make a reservation. To keep things simple, we assume that sailors make transactions using a psql interface.

Rusty likes the boat Clipper and would like to use it from 13 April 2008 to 15 April 2008. He types

```
SELECT r.startDate, r.endDate
FROM   reservation r
WHERE  r.boatName = 'Clipper' AND
       r.startDate <= '15-Apr-08' AND
       r.endDate >= '13-Apr-08';
```

What is the intuitive meaning of the query?
What does Rusty get to see?

Transactions (Example ctd)

Rusty decides to reserve Clipper from 13 April to 15 April and types

```
INSERT INTO reservation
VALUES ('Clipper', '13-Apr-05', '15-Apr-05', 'Rusty');
```

Lubber wants to reserve a boat from 12 April 2005 to 14 April 2005 and types

```
      SELECT r.boatName
      FROM   reservation r
      WHERE  r.startDate <= '14-Apr-05' AND
             r.endDate >= '12-Apr-05';
```

What is the intuitive meaning of the query?
What does Lubber get to see?

Transactions (Example ctd)

Lubber also likes Clipper and types

```
INSERT INTO reservation VALUES
   ('Clipper', '12-Apr-05', '14-Apr-05', 'Lubber');
```

What happens?

Both, Rusty and Lubber type commit.

What happens?

Transactions (Example ctd)

The manager of the club connects to the database and types the following query:

```
SELECT r1.boatName, r1.startDate, r1.endDate,
       r2.startDate, r2.endDate
FROM   reservation r1, reservation r2
WHERE  r1.boatName = r2.boatName AND
       r1.startDate <= r2.endDate AND
       r2.startDate <= r1.endDate AND
       r1.startDate <= r2.startDate;
```

What is the intuitive meaning of the query?
What does the manager get to see?
Has the isolation property of transactions been guaranteed?

SQL: Summary

- SQL combines DDL and DML
- DDL implements basic concepts of relational data model (domains, relations, schemas, integrity constraints)
- The core DML (w/o) aggregation is essentially equivalent to first order predicate logic
- The DML has the same expressivity as relational algebra
- Aspects of both, predicate logic and relational algebra, are present in the SQL query language
- Further aspects include generic integrity constraints, views, and transactions

References

In preparing the lectures I have used several sources.
The main ones are the following:

Books:

- A First Course in Database Systems, by J. Ullman and J. Widom
- Fundamentals of Database Systems, by R. Elmasri and S. Navathe

Slides:

- The slides of this chapter are to a large part translations of material prepared by Maurizio Lenzerini (University of Rome, “La Sapienza”) and Diego Calvanese (Free University of Bozen-Bolzano) for their introductory course on databases at the University of Rome, “La Sapienza”