Basi di Dati e Sistemi Informativi I

SQL

SQL queries

SQL as a query language

- SQL expresses queries in declarative way
  - queries specify the properties of the result, not the way to obtain it
- Queries are translated by the query optimizer into the procedural language internal to the DBMS
- The programmer should focus on readability, not on efficiency

SQL queries

- SQL queries are expressed by the select statement
- Syntax:
  `select` *AttrExpr* [[`as`] *Alias*] {, *AttrExpr* [[`as`] *Alias*] }
  `from` *Table* [[`as`] *Alias*] {, [[`as`] *Alias*] }
  [`where` *Condition*]
- The three parts of the query are usually called:
  - target list
  - `from` clause
  - `where` clause
- The query considers the cartesian product of the tables in the `from` clause, considers only the rows that satisfy the condition in the `where` clause and for each row evaluates the attribute expressions in the target list

Example database

EMPLOYEE

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

DEPARTMENT

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San José

Simple SQL query

- Find the salaries of employees named Brown:

```
select Salary as Remuneration
from Employee
where Surname = 'Brown'
```

- Result:

Remuneration
45
80

* in the target list

- Find all the information relating to employees named Brown:

```
select *
from Employee
where Surname = 'Brown'
```

- Result:

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	Brown	Planning	14	80	London

Attribute expressions

- Find the monthly salary of the employees named White:

```
select Salary / 12 as MonthlySalary
from Employee
where Surname = 'White'
```

- Result:

MonthlySalary
3.00

Simple join query

- Find the names of the employees and the cities in which they work:

```
select Employee.FirstName, Employee.Surname,
       Department.City
from Employee, Department
where Employee.Dept = Department.DeptName
```

- Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

Table aliases

- Find the names of the employees and the cities in which they work (using an alias):

```
select FirstName, Surname, D.City
from Employee, Department D
where Dept = DeptName
```

- Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

Predicate conjunction

- Find the first names and surnames of the employees who work in office number 20 of the Administration department:

```
select FirstName, Surname
from Employee
where Office = '20' and
      Dept = 'Administration'
```

- Result:

FirstName	Surname
Gus	Green

Predicate disjunction

- Find the first names and surnames of the employees who work in either the Administration or the Production department:

```
select FirstName, Surname
from Employee
where Dept = 'Administration' or
      Dept = 'Production'
```

- Result:

FirstName	Surname
Mary	Brown
Charles	White
Gus	Green
Pauline	Bradshaw
Alice	Jackson

Complex logical expression

- Find the first names of the employees named Brown who work in the Administration department or the Production department:

```
select FirstName
from Employee
where Surname = 'Brown' and
      (Dept = 'Administration' or
       Dept = 'Production')
```

- Result:

FirstName
Mary

Operator `like`

- Find the employees with surnames that have ‘r’ as the second letter and end in ‘n’:

```
select *
from Employee
where Surname like '_r%n'
```

- _ : arbitrary character;
- % : a string (possibly empty) of arbitrary characters
- Result:

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Gus	Green	Administration	20	40	Oxford
Charles	Brown	Planning	14	80	London

Management of null values

- Null values may mean that:
  - a value is not applicable
  - a value is applicable but unknown
  - it is unknown if a value is applicable or not
- SQL-89 uses a two-valued logic
  - a comparison with *null* returns FALSE
- SQL-2 uses a three-valued logic
  - a comparison with *null* returns UNKNOWN
- To test for null values:

  Attribute `is` [`not`] `null`

Algebraic interpretation of SQL queries

- The generic query:

```
select T_1.Attribute_11, …, T_h.Attribute_hm
from Table_1 T_1, …, Table_n T_n
where Condition
```

- corresponds to the relational algebra query:

$$\pi_{T\_1.Attribute\_11,\ldots,T\_h.Attribute\_hm} (\sigma_{Condition} (Table\_1 \bowtie \ldots \bowtie Table\_n))$$

Duplicates

- In relational algebra and calculus the results of queries do not contain duplicates
- In SQL, tables may have identical rows
- Duplicates can be removed using the keyword `distinct`

```
select City
from Department
```

```
select distinct City
from Department
```

City
London
Toulouse
Brighton
London
San José

City
London
Toulouse
Brighton
San José

Joins in SQL-2

- SQL-2 introduced an alternative syntax for the representation of joins, representing them explicitly in the `from` clause:

  `select` *AttrExpr* [[`as`] *Alias*] {, *AttrExpr* [[`as`] *Alias*] }
  `from` *Table* [[`as`] *Alias*]
  { [*JoinType*] `join` *Table* [[`as`] *Alias*] `on` *JoinConditions* }
  [`where` *OtherCondition*]

- *JoinType* can be any of `inner`, `right` [`outer`], `left` [`outer`] or `full` [`outer`], permitting the representation of outer joins
- The keyword `natural` may precede *JoinType* (rarely implemented)

Inner join in SQL-2

- Find the names of the employees and the cities in which they work:

```
select FirstName, Surname, D.City
from Employee inner join Department as D
     on Dept = DeptName
```

- Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

Example database, drivers and cars

DRIVER

FirstName	Surname	DriverID
Mary	Brown	VR 2030020Y
Charles	White	PZ 1012436B
Marco	Neri	AP 4544442R

AUTOMOBILE

CarRegNo	Make	Model	DriverID
ABC 123	BMW	323	VR 2030020Y
DEF 456	BMW	Z3	VR 2030020Y
GHI 789	Lancia	Delta	PZ 1012436B
BBB 421	BMW	316	MI 2020030U

Left join

- Find the drivers with their cars, including the drivers without cars:

```
select FirstName, Surname, Driver.DriverID
       CarRegNo, Make, Model
from Driver left join Automobile on
     (Driver.DriverID = Automobile.DriverID)
```

- Result:

FirstName	Surname	DriverID	CarRegNo	Make	Model
Mary	Brown	VR 2030020Y	ABC 123	BMW	323
Mary	Brown	VR 2030020Y	DEF 456	BMW	Z3
Charles	White	PZ 1012436B	GHI 789	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL

Full join

- Find all the drivers and all the cars, showing the possible relationships between them:

```
select FirstName, Surname, Driver.DriverID
       CarRegNo, Make, Model
from Driver full join Automobile on
     (Driver.DriverID = Automobile.DriverID)
```

- Result:

FirstName	Surname	DriverID	CarRegNo	Make	Model
Mary	Brown	VR 2030020Y	ABC 123	BMW	323
Mary	Brown	VR 2030020Y	DEF 456	BMW	Z3
Charles	White	PZ 1012436B	GHI 789	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL
NULL	NULL	NULL	BBB 421	BMW	316

Table variables

- Table aliases may be interpreted as table variables
- They correspond to the renaming operator ρ of relational algebra
- Find all the same surname (but different first names) of an employee belonging to the Administration department:

```
select E1.FirstName, E1.Surname
from Employee E1, Employee E2
where E1.Surname = E2.Surname and
      E1.FirstName <> E2.FirstName and
      E2.Dept = 'Administration'
```

- Result:

FirstName	Surname
Charles	Brown

Ordering

- The `order by` clause, at the end of the query, orders the rows of the result; syntax:

  `order by` *OrderingAttribute* [`asc` | `desc`]
  {, *OrderingAttribute* [`asc` | `desc`] }
- Extract the content of the AUTOMOBILE table in descending order of make and model:

```
select *
from Automobile
order by Make desc, Model desc
```

- Result:

CarRegNo	Make	Model	DriverID
GHI 789	Lancia	Delta	PZ 1012436B
DEF 456	BMW	Z3	VR 2030020Y
ABC 123	BMW	323	VR 2030020Y
BBB 421	BMW	316	MI 2020030U

Aggregate queries

- Aggregate queries cannot be represented in relational algebra
- The result of an aggregate query depends on the consideration of sets of rows
- SQL-2 offers five aggregate operators:
  - `count`
  - `sum`
  - `max`
  - `min`
  - `avg`

Operator `count`

- `count` returns the number of rows or distinct values; syntax:

  `count(< * | [ distinct | all ]` *AttributeList* `>)`

  (*) returns the number os rows; `distinct` returns the the number of different values for the *AttributeList;* `all` returns the number of rows that possess values different from the null value for the *AttributeList.*
- Find the number of employees:

```
select count(*)
from Employee
```

- Find the number of different values on the attribute Salary for all the rows in EMPLOYEE:

```
select count(distinct Salary)
from Employee
```

- Find the number of rows of EMPLOYEE having a not null value on the attribute Salary:

```
select count(all Salary)
from Employee
```


Sum, average, maximum and minimum

- Syntax:
  `< sum | max | min | avg > ([ distinct | all ]` *AttributeExpr* `)`
- `Distinct` ignores duplicates, while `all` leaves out only null values
- Find the sum of the salaries of the Administration department:

```
select sum(Salary) as SumSalary
from Employee
where Dept = 'Administration'
```

- Result:

SumSalary
125

Aggregate queries with join

- Find the maximum salary among the employees who work in a department based in London:

```
select max(Salary) as MaxLondonSal
from Employee, Department
where Dept = DeptName and
      Department.City = 'London'
```

- Result:

MaxLondonSal
80

Aggregate queries and target list

- Incorrect query:

```
select FirstName, Surname, max(Salary)
from Employee, Department
where Dept = DeptName and
      Department.City = 'London'
```

- Whose name? The target list must be homogeneous

- Find the maximum and minimum salaries of all employees:

```
select max(Salary) as MaxSal,
       min(Salary) as MinSal
from Employee
```

- Result:

MaxSal	MinSal
80	36

Group by queries

- Queries may apply aggregate operators to subsets of rows
- Find the sum of salaries of all the employees of the same department:

```
select Dept, sum(Salary)as TotSal
from Employee
group by Dept
```

- Result:

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

Semantics of group by queries, 1

- First, the query is executed without `group by` and without aggregate operators:

```
select Dept, Salary
from Employee
```

Dept	Salary
Administration	45
Production	36
Administration	40
Distribution	45
Planning	80
Planning	73
Administration	40
Production	46

Semantics of group by queries, 2

- … then the query result is divided in subsets characterized by the same values for the attributes appearing as argument of the `group by` clause (in this case attribute Dept):
- Finally, the aggregate operator is applied separately to each subset

Dept	Salary
Administration	45
Administration	40
Administration	40
Distribution	45
Planning	80
Planning	73
Production	36
Production	46

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

Group by queries and target list

- Incorrect query:

```
select Office
from Employee
group by Dept
```

- This query is incorrect, in that a number of values of the Office attribute will correspond to each value of the Dept attribute. Instead, after the grouping has been carried out, each sub-group of rows must correspond to a single row in the table resulting from the query.

Group by queries and target list

- Incorrect query:

```
select DeptName, count(*), D.City
from Employee E join Department D
     on (E.Dept = D.DeptName)
group by DeptName
```

- This query should return the department name, the number of employees of each department, and the city in which the department is based. Given that the attribute DeptName is the key of DEPARTMENT, a particular value of City corresponds to each value of DetpName. The system could therefore provide a correct response, but SQL prohibits queries of this nature.

Group by queries and target list

- Correct query:

```
select DeptName, count(*), D.City
from Employee E join Department D
     on (E.Dept = D.DeptName)
group by DeptName, D.City
```

Group predicates

- When conditions are on the result of an aggregate operator, it is necessary to use the `having` clause
- Find which departments spend more than 100 on salaries:

```
select Dept
from Employee
group by Dept
having sum(Salary) > 100
```

- Result:

Dept
Administration
Planning

`where` or `having`?

- Only predicates containing aggregate operators should appear in the argument of the `having` clause

- Find the departments in which the average salary of employees working in office number 20 is higher than 25:

```
select Dept
from Employee
where Office = '20'
group by Dept
having avg(Salary) > 25
```

Syntax of an SQL query

- Considering all the described clauses, the syntax is:

```
select TargetList
from TableList
[ where Condition ]
[ group by GroupingAttributeList ]
[ having AggregateCondition ]
[ order by OrderingAttributeList ]
```


Set queries

- SQL also provides set operators, similar to those we saw in relation algebra: `union`, `intersect`, `except` or `minus`
- Syntax:

SelectSQL { < `union` | `intersect` | `except` > [`all`] *SelectSQL* }

- Find the first names and surnames of the employees:

```
select FirstName as Name
from Employee
 union
select Surname
from Employee
```

- Duplicates are removed (unless the `all` option is used)

Intersection

- Find the surnames of employees that are also first names:

```
select FirstName as Name
from Employee
 intersect
select Surname
from Employee
```

- equivalent to:

```
select E1.FirstName as Name
from Employee E1, Employee E2
where E1.FirstName = E2.Surname
```


Difference

- Find the surnames of employees that are not also first names:

```
select FirstName as Name
from Employee
 except
select Surname
from Employee
```

- Can be represented with a nested query (see later)

Nested queries

- In the `where` clause may appear predicates that:
  - compare an attribute (or attribute expression) with the result of an SQL query; syntax:
    ScalarValue Operator < `any` | `all` > *SelectSQL*
    - `any`: the predicate is true if at least one row returned by *SelectSQL* satisfies the comparison
    - `all`: the predicate is true if all the rows returned by *SelectSQL* satisfy the comparison
  - use the existential quantifier on an SQL query; syntax:
    `exists` *SelectSQL*
    - the predicate is true if *SelectSQL* returns a non-empty result
- The query appearing in the `where` clause is called nested query

Simple nested queries, 1

- Find the employees who work in departments in London:

```
select FirstName, Surname
from Employee
where Dept = any (select DeptName
                  from Department
                  where City = 'London')
```

- Equivalent to (without nested query):

```
select FirstName, Surname
from Employee, Department D
where Dept = DeptName and
      D.City = 'London'
```

Simple nested queries, 2

- Find the employees of the Planning department, having the same first name as a member of the Production department:
  - without nested queries:

```
select E1.FirstName, E1.Surname
from Employee E1, Employee E2
where E1. FirstName = E2.FirstName and
      E2.Dept = 'Production' and
      E1.Dept = 'Planning'
```

  - with a nested query:

```
select FirstName, Surname
from Employee
where Dept = 'Planning' and
      FirstName = any
        (select FirstName
         from Employee
         where Dept = 'Production')
```


Negation with nested queries

- Find the departments in which there is no one named Brown:

```
select DeptName
from Department
where DeptName <> all (select Dept
                        from Employee
                        where Surname = 'Brown')
```

- Alternatively:

```
select DeptName
from Department
   except
select Dept
from Employee
where Surname = 'Brown'
```

Operators `in` and `not in`

- Operator `in` is a shorthand for `= any`

```
select FirstName, Surname
from Employee
where Dept in (select DeptName
               from Department
               where City = 'London')
```

- Operator `not in` is a shorthand for `<> all`

```
select DeptName
from Department
where DeptName not in (select Dept
                       from Employee
                       where Surname = 'Brown')
```


`max` and `min` with a nested query

- Queries using the aggregate operators `max` and `min` can be expressed with nested queries
- Find the department of the employee earning the highest salary
  - with `max`:

```
select Dept
from Employee
where Salary in (select max(Salary)
                 from Employee
```

  - with a nested query:

```
select Dept
from Employee
where Salary >= all (select Salary
                     from Employee
```

Complex nested queries, 1

- With regard to the visibility (or *scope*) of SQL variables, there is a restriction on the use of a variable: a variable can be used only within the query where in which it is defined or within a query that recursively nested in the query where it is defined. The nested query may use variables of the external query ('transfer of bindings')
- Incorrect query:

```
select *
from Employee
where Dept in
   (select DeptName
         from Department D1
         where DeptName = 'Production') or
      Dept in (select DeptName
             from Department D2
             where D2.City = D1.City)
```

- The query is incorrect because variable `D1` is not visible in the second nested query

Complex nested queries, 2

- Semantics: the nested query is evaluated for each row of the external query
- Find all the homonyms, i.e., persons who have the same first name and surname, but different tax codes:

```
select *
from Person P
where exists (select *
              from Person P1
              where P1.FirstName = P.FirstName
                and P1.Surname = P.Surname
                and P1.TaxCode <> P.TaxCode)
```

Complex nested queries, 3

- Find all the persons who do not have homonyms:

```
select *
from Person P
where not exists
        (select *
         from Person P1
         where P1.FirstName = P.FirstName
           and P1.Surname = P.Surname
           and P1.TaxCode <> P.TaxCode)
```


Tuple constructor

- The comparison with the nested query may involve more than one attribute
- The attributes must be enclosed within a pair of curved brackets (tuple constructor)
- The previous query can be expressed in this way:

```
select *
from Person P
where (FirstName,Surname) not in
        (select FirstName, Surname
         from Person P1
         where P1.TaxCode <> P.TaxCode)
```

Comments on nested queries

- The use of nested queries may produce ‘less declarative’ queries, but they often improve readability
- Complex queries can become very difficult to understand
- The use of variables must respect visibility rules
  - a variable can be used only within the query where it is defined or within a query that is recursively nested in the query where it is defined