



**SQL**

# SQL

- The name is an acronym for *Structured Query Language*
- Far richer than a query language: both a DML and a DDL
- History:
  - First proposal: SEQUEL (IBM Research, 1974)
  - First implementation in SQL/DS (IBM, 1981)
- Initial “standardization” crucial for its diffusion
  - Since 1983, *standard de facto*
  - First standard, 1986, revised in 1989 (SQL-89)
  - Second standard, 1992 (SQL-2 or SQL-92)
  - Third standard, 1999 (SQL-3 or SQL-99)
  - Etc.
- Most relational systems support the base functionality of the standard and offer proprietary extensions



# Domains

- Domains specify the content of attributes
- Two categories
  - Elementary (predefined by the standard)
  - User-defined

## Examples elementary domains

- Character
  - Single characters or strings
  - Strings may be of variable length
  - A Character set different from the default one can be used (e.g., Latin, Greek, Cyrillic, etc.)
  - Syntax:

```
character [varying ][(Length)]  
          [character set CharSetName]
```
  - It is possible to use `char` and `varchar`, respectively for `character` and `character varying`

## Examples elementary domains (cont.d)

- Bit
  - Single boolean values or strings of boolean values (may be variable in length)
  - Syntax:  
`bit [varying ] [ (Length) ]`
- Exact numeric domains
  - Exact values, integer or with a fractional part
  - Different alternatives:  
`numeric [ ( Precision [, Scale ] ) ]`  
`decimal [ ( Precision [, Scale ] ) ]`  
`integer`  
`smallint`

## Examples elementary domains (cont.d)

- Approximate numeric domains
  - Approximate real values
  - Based on a floating point representation

```
float [ (Precision) ]  
double precision  
real
```

## Examples elementary domains (cont.d)

- Temporal instants

date

time [ ( *Precision* ) [with time zone ]

timestamp [ ( *Precision* ) [with time zone ]

- Temporal intervals

interval *FirstUnitOfTime* [ to *LastUnitOfTime* ]

- Units of time are divided into two groups:

- year, month
- day, hour, minute, second

# Schema definition

- A schema is a collection of objects:
  - domains, tables, indexes, assertions, views, privileges
- A schema has a name and an owner (the authorization)
- Syntax:

```
create schema [ SchemaName ]  
[[ authorization ] Authorization ]  
    { SchemaElementDefinition }
```



# Table definition

- An SQL table consists of
  - an ordered set of attributes
  - a (possibly empty) set of constraints
- Statement `create table`
  - defines a relation schema, creating an empty instance

- Syntax:

```
create table TableName
(
  AttributeName Domain [ DefaultValue ] [ Constraints ]
  {, AttributeName Domain [ DefaultValue ] [ Constraints ] }
  [ OtherConstraints ]
)
```

## Example of create table

```
create table Employee
(
    RegNo          character(6) primary key,
    FirstName      character(20) not null,
    Surname        character(20) not null,
    Dept           character (15)
                references Department(DeptName)
                on delete set null
                on update cascade,
    Salary         numeric(9) default 0,
    City           character(15),
    unique(Surname,FirstName)
)
```

# User defined domains

- Comparable to the definition of variable types in programming languages
- A domain is characterized by
  - name
  - elementary domain
  - default value
  - set of constraints
- Syntax:

```
create domain DomainName as ElementaryDomain  
[ DefaultValue ] [ Constraints ]
```
- Example:

```
create domain smallintNULL as smallint default null
```

## Default domain values

- Define the value that the attribute must assume when a value is not specified during row insertion
- Syntax:

`default < GenericValue | user | null >`

- *GenericValue* represents a value compatible with the domain, in the form of a constant or an expression
- `user` is the login name of the user who issues the command

# Intra-relational constraints

- Constraints are conditions that must be verified by every database instance
- Intra-relational constraints involve a single relation
  - `not null` (on single attributes)
  - `unique`: permits the definition of keys; syntax:
    - for single attributes:  
`unique`, after the domain
    - for multiple attributes:  
`unique( Attribute {, Attribute } )`
  - `primary key`: defines the primary key (once for each table; implies `not null`); syntax like `unique`
  - `check`: described later

## Example of intra-relational constraints

- Each pair of FirstName and Surname uniquely identifies each element

```
FirstName character(20) not null,  
Surname character(20) not null,  
unique(FirstName, Surname)
```

- Note the difference with the following (stricter) definition:

```
FirstName character(20) not null unique,  
Surname character(20) not null unique,
```

# Inter-relational constraints

- Constraints may take into account several relations
  - check: described later
  - `references` and `foreign key` permit the definition of referential integrity constraints; syntax:
    - for single attributes  
`references` after the domain
    - for multiple attributes  
`foreign key ( Attribute {, Attribute } )`  
`references ...`
  - It is possible to associate reaction policies to violations of referential integrity

# Reaction policies for referential integrity constraints

- Reactions operate on the internal table, after changes to the external table
- Violations may be introduced (1) by updates on the referred attribute or (2) by row deletions
- Reactions:
  - cascade: propagate the change
  - set null: nullify the referring attribute
  - set default: assign the default value to the referring attribute
  - no action (restrict): forbid the change on the external table
- Reactions may depend on the event; syntax:  
on <delete | update >  
    <cascade | set null | set default | no action >



## Example of inter-relational constraint and reaction policies

```
create table Employee
(
    RegNo          char(6),
    FirstName      char(20) not null,
    Surname        char(20) not null,
    Dept           char(15),
    Salary         numeric(9) default 0,
    City           char(15),
    primary key(RegNo),
    foreign key(Dept)
        references Department(DeptName)
        on delete set null
        on update cascade,
    unique(FirstName, Surname)
)
```

# Schema updates

- Two SQL statements:
  - alter (alter domain ..., alter table ...)
  - drop
    - drop < schema | domain | table | view | assertion >  
*ComponentName* [ restrict | cascade ]
- Examples:
  - alter table Department  
add column NoOfOffices numeric(4)
  - drop table TempTable cascade

## 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

NOTE: “as Remuneration” is optional

## \* 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 with surname 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'
```

- 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
- Alternatives:
  - Two-valued logic: a comparison with *null* returns FALSE
  - 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  $T_1.Attribute_{11}, \dots, T_h.Attribute_{hm}$   
from  $Table_1 T_1, \dots, Table_n T_n$   
where  $Condition$ 
```

- corresponds to the relational algebra query:

$$\pi_{T_1.Attribute_{11}, \dots, T_h.Attribute_{hm}} (\sigma_{Condition} (Table_1 \times \dots \times Table_n))$$



# Duplicates

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

```
select City  
from Department
```

City
London
Toulouse
Brighton
London
San José

```
select distinct City  
from Department
```

City
London
Toulouse
Brighton
San José

# Joins in SQL

- 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*
- NOTE: All our examples will use the simpler syntax for joins (seen before)

## Table variables

- Table aliases may be interpreted as table variables
- They correspond to the renaming operator  $\rho$  of relational algebra
- Find the first names and surnames of employees that have the same surname but different first name as employees 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

# 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

# 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

- The result of an aggregate query may depend on a calculation and/or on the evaluation of a predicate on sets of rows rather than just on individual rows
- SQL offers five main aggregate operators:
  - count
  - sum
  - max
  - min
  - avg

# Operator count

- `count` returns the number of rows or distinct values; syntax:  
`count(< * | [distinct | all ] 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 )`

- 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
```

- Incorrect query:

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

- Correct query:

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

NOTE: the GROUP by clause might be implemented slightly differently in different DBs.

## 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

- A single select cannot represent unions
- Syntax:

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

- Find the list of 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 Surname as Name
from Employee
except
select FirstName
from Employee
```

- Also called MINUS
- Not available in MySQL
- 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:  
$$\text{ScalarValue Operator} < \text{any} \mid \text{all} > \text{SelectSQL}$$
    - `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:  
$$\text{exists SelectSQL}$$
    - the predicate is true if *SelectSQL* returns a non-empty result
- The query appearing in the `where` clause is called nested query
- IN same as “= any”, NOT IN same as “<> ALL”

## Simple nested queries, 1

- Find the firstname and surname of 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 firstname and surname of 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'
```



## **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 = any (select max(Salary)
                    from Employee)
```

- with a nested query:

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

# Data modification in SQL

- Statements for
  - insertion (`insert`)
  - deletion (`delete`)
  - change of attribute values (`update`)
- All the statements can operate on a set of tuples (set-oriented)
- In the condition it is possible to access other relations

# Insertions, 1

- Syntax:

```
insert into TableName [ (AttributeList) ]  
    <values (ListOfValues) | SelectSQL>
```

- Using values:

```
insert into Department (DeptName, City)  
    values ( 'Production' , 'Toulouse' )
```

- Using a subquery:

```
insert into LondonProducts  
    (select Code, Description  
     from Product  
     where ProdArea = 'London' )
```

## Insertions, 2

- The ordering of the attributes (if present) and of values is meaningful (first value with the first attribute, and so on)
- If *AttributeList* is omitted, all the relation attributes are considered, in the order in which they appear in the table definition
- If *AttributeList* does not contain all the relation attributes, to the remaining attributes it is assigned the default value (if defined) or the null value

# Deletions, 1

- Syntax:

```
delete from TableName [ where Condition ]
```

- Remove the Production department:

```
delete from Department  
where DeptName = 'Production'
```

- Remove the departments without employees:

```
delete from Department  
where DeptName <> all (select Dept  
                        from Employee)
```

## Deletions, 2

- The `delete` statement removes from the table all the tuples that satisfy the condition
- The removal may produce deletions from other tables if a referential integrity constraint with `cascade` policy has been defined
- If the `where` clause is omitted, `delete` removes all the tuples
- To remove all the tuples from DEPARTMENT (keeping the table schema):

```
delete from Department
```

- To remove table DEPARTMENT completely (content and schema):

```
drop table Department cascade
```

# Updates, 1

- Syntax:

```
update TableName
  set Attribute = < Expression | SelectSQL | null | default >
  {, Attribute = < Expression | SelectSQL | null | default >}
  [where Condition]
```

- Examples:

```
update Employee
  set Salary = Salary + 5
  where RegNo = 'M2047'
```

```
update Employee
  set Salary = Salary * 1.1
  where Dept = 'Administration'
```

## Updates, 2

- Since the language is set oriented, the order of the statements is important

```
update Employee
  set Salary = Salary * 1.1
  where Salary <= 30
```

```
update Employee
  set Salary = Salary * 1.15
  where Salary > 30
```

- If the statements are issued in this order, some employees may get a double raise



# Generic integrity constraints

- The `check` clause can be used to express arbitrary constraints during schema definition
- Syntax:  
`check (Condition)`
- *Condition* is what can appear in a `where` clause (including nested queries)
- E.g., the definition of an attribute `Superior` in the schema of table `EMPLOYEE`:  
`Superior character(6)`  
`check (Superior like "1%")`
- E.g., referential integrity:  
`Dept character (15) check`  
`(Dept = any (select DeptName from Department))`

# Assertions

- Assertions permit the definition of constraints outside of table definitions
- Useful in many situations (e.g., to express generic inter-relational constraints)
- An assertion associates a name to a `check` clause; syntax:

```
create assertion AssertionName check (Condition)
```

- There must always be at least one tuple in table EMPLOYEE:

```
create assertion AlwaysOneEmployee  
  check (1 <= (select count(*)  
               from Employee))
```

# Views

- Syntax:

```
create view ViewName [ (AttributeList) ] as SelectSQL  
    [ with [ local | cascaded ] check option ]
```

```
create view AdminEmployee  
    (RegNo, FirstName, Surname, Salary) as  
select RegNo, FirstName, Surname, Salary  
from Employee  
where Dept = 'Administration' and Salary > 10
```

```
create view JuniorAdminEmployee as  
select *  
from AdminEmployee  
where Salary < 50  
with check option
```

NOTE: check option used for views that can be modified (after the modification the rows should still belong to the view)

# Embedded SQL

- Traditional applications often need to “embed” SQL statements inside the instructions of a procedural programming language (C, Java, etc.)
- Programs with embedded SQL use a precompiler to manage SQL statements
- Embedded statements are preceded by ‘\$’ or ‘EXEC SQL’
- Program variables may be used as parameters in the SQL statements (preceded by ‘:’)
- `select` producing a single row and update commands may be embedded easily
- The SQL environment offers a predefined variable `sqlcode` which describes the status of the execution of the SQL statements (zero if the SQL statement executed successfully)