# Week -2

# Introduction to SQL

## Overview of the SQL Query Language

The SQL language has several parts:

- **Data-definition language** (DDL). The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Data-manipulation language** (DML). The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity**. The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition**. The SQL DDL includes commands for defining views.
- **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL and dynamic SQL. Embedded and dynamic SQL define** how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

### SQL Data Definition

The set of relations in a database must be specified to the system by means of a data-definition language (DDL).

The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation.
- The types of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

### Domain Types in SQL

The SQL standard supports a variety of built-in types, including:

- **char($n$)**: A fixed-length character string with user-specified length $n$. The full form, **character**, can be used instead.
- **varchar($n$)**: A variable-length character string with user-specified maximum length $n$. The full form, **character varying**, is equivalent.
- **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.
- **smallint**: A small integer (a machine-dependent subset of the integer type).
- **numeric($p$, $d$)**: A fixed-point number with user-specified precision. The number consists of $p$ digits (plus a sign), and $d$ of the $p$ digits are to the right of the decimal point. Thus, **numeric**(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float($n$)**: A floating-point number, with precision of at least $n$ digits.

## Basic Schema Definition

We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database.

```
create table department
        (dept_name  varchar (20),
         building   varchar (15),
         budget     numeric (12,2),
         primary key (dept_name));
```

The general form of the **create table** command is:

```
create table r
        (A1  D1,
        A2  D2,
        ...,
        An  Dn,
        ⟨integrity-constraint1⟩,
        ...,
        ⟨integrity-constraintk⟩);
```

A newly created relation is empty initially. We can use the **insert** command to load data into the relation. For example, if we wish to insert the fact that there is an instructor named Smith in the Biology department with *instructor_id* 10211 and a salary of $66,000, we write:

```
insert into instructor
        values (10211, 'Smith', 'Biology', 66000);
```

The values are specified in the order in which the corresponding attributes are listed in the relation schema.

### Some More Examples

1.
```
create table department
    (dept_name      varchar (20),
     building       varchar (15),
     budget         numeric (12,2),
     primary key (dept_name));
```

2.
```
create table course
    (course_id      varchar (7),
     title          varchar (50),
     dept_name      varchar (20),
     credits        numeric (2,0),
     primary key (course_id),
     foreign key (dept_name) references department);
```

3.
```
create table instructor
    (ID             varchar (5),
     name           varchar (20) not null,
     dept_name      varchar (20),
     salary         numeric (8,2),
     primary key (ID),
     foreign key (dept_name) references department);
```

4.
```
create table section
    (course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     building       varchar (15),
     room_number    varchar (7),
     time_slot_id   varchar (4),
     primary key (course_id, sec_id, semester, year),
     foreign key (course_id) references course);
```

5.
```
create table teaches
    (ID             varchar (5),
     course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6),
     year           numeric (4,0),
     primary key (ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section,
     foreign key (ID) references instructor);
```

To remove a relation from an SQL database, we use the **drop table** command. The **drop table** command deletes all information about the dropped relation from the database. The command

**drop table** *r*;

is a more drastic action than

**delete from** *r*;

The latter retains relation *r*, but deletes all tuples in *r*. The former deletes not only all tuples of *r*, but also the schema for *r*. After *r* is dropped, no tuples can be inserted into *r* unless it is re-created with the **create table** command.

We use the **alter table** command to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the **alter table** command is

**alter table** *r* **add** *A D*;

where *r* is the name of an existing relation, *A* is the name of the attribute to be added, and *D* is the type of the added attribute. We can drop attributes from a relation by the command

**alter table** *r* **drop** *A*;

where *r* is the name of an existing relation, and *A* is the name of an attribute of the relation. Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.

## Basic Structure of SQL Queries

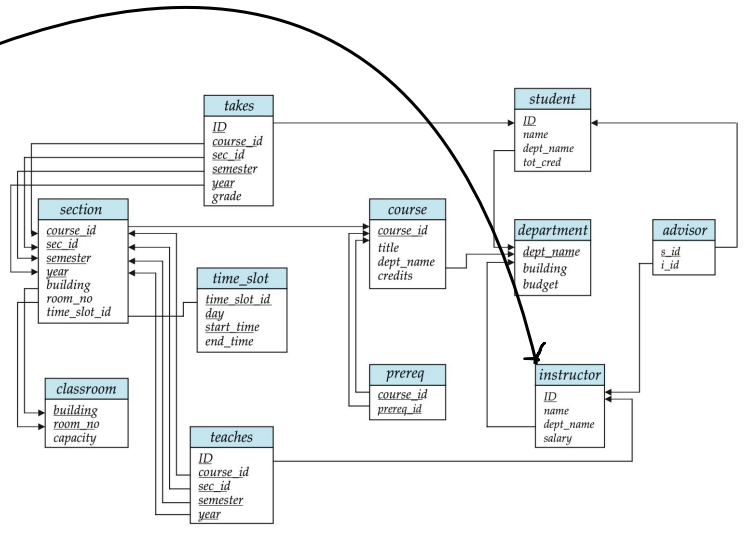
The basic structure of an SQL query consists of three clauses: **select, from, and where.**

### Queries on a Single Relation

Let us consider a simple query using our university example, "Find the names of all instructors." Instructor names are found in the *instructor* relation, so we put that relation in the **from** clause. The instructor's name appears in the *name* attribute, so we put that in the **select** clause.

| *name* |
|---|
| Srinivasan |
| Wu |
| Mozart |
| Einstein |
| El Said |
| Gold |
| Katz |
| Califieri |
| Singh |
| Crick |
| Brandt |
| Kim |

**select** *name*
**from** *instructor*;



Now consider another query, "Find the department names of all instructors," which can be written as:

```
select distinct dept_name
from instructor;
```

| dept_name |
|---|
| Comp. Sci. |
| Finance |
| Music |
| Physics |
| History |
| Physics |
| Comp. Sci. |
| History |
| Finance |
| Biology |
| Comp. Sci. |
| Elec. Eng. |

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

```
select all dept_name
from instructor;
```

The **select** clause may also contain arithmetic expressions involving the operators +, −, ∗, and / operating on constants or attributes of tuples. For example, the query:

```
select ID, name, dept_name, salary * 1.1
from instructor;
```

Result of "Find the names of all instructors in the Computer Science department who have salary greater than $70,000."

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 70000;
```

| name |
|---|
| Katz |
| Brandt |

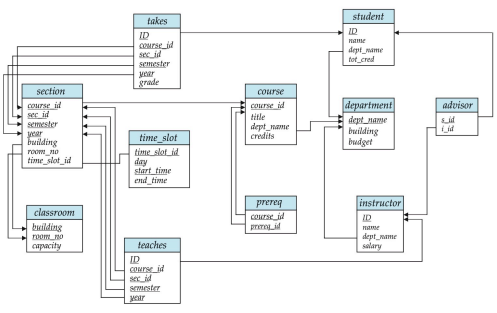
## Queries on Multiple Relations

An an example, suppose we want to answer the query "Retrieve the names of all instructors, along with their department names and department building name."

Looking at the schema of the relation *instructor*, we realize that we can get the department name from the attribute *dept_name*, but the department building name is present in the attribute *building* of the relation *department*. To answer the query, each tuple in the *instructor* relation must be matched with the tuple in the *department* relation whose *dept_name* value matches the *dept_name* value of the *instructor* tuple.

In SQL, to answer the above query, we list the relations that need to be accessed in the **from** clause, and specify the matching condition in the **where** clause. The above query can be written in SQL as

```
select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name= department.dept_name;
```



| *name* | *dept_name* | *building* |
|---|---|---|
| Srinivasan | Comp. Sci. | Taylor |
| Wu | Finance | Painter |
| Mozart | Music | Packard |
| Einstein | Physics | Watson |
| El Said | History | Painter |
| Gold | Physics | Watson |
| Katz | Comp. Sci. | Taylor |
| Califieri | History | Painter |
| Singh | Finance | Painter |
| Crick | Biology | Watson |
| Brandt | Comp. Sci. | Taylor |
| Kim | Elec. Eng. | Taylor |

The result of "Retrieve the names of all instructors, along with their department names and department building name."

We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the **select** clause, the **from** clause, and the **where** clause. The role of each clause is as follows:

- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the **from** clause.

A typical SQL query has the form

**select** $A_1, A_2, \ldots, A_n$
**from** $r_1, r_2, \ldots, r_m$
**where** $P$;

Each $A_i$ represents an attribute, and each $r_i$ a relation. $P$ is a predicate. If the **where** clause is omitted, the predicate $P$ is **true**.

Although the clauses must be written in the order **select**, **from**, **where**, the easiest way to understand the operations specified by the query is to consider the clauses in operational order: first **from**, then **where**, and then **select**.[1]

The **from** clause by itself defines a Cartesian product of the relations listed in the clause. It is defined formally in terms of set theory, but is perhaps best understood as an iterative process that generates tuples for the result relation of the **from** clause.

```
for each tuple t1 in relation r1
        for each tuple t2 in relation r2
                ...
                for each tuple tm in relation rm
                        Concatenate t1, t2, ..., tm into a single tuple t
                        Add t into the result relation
```

The result relation has all attributes from all the relations in the **from** clause. Since the same attribute name may appear in both $r_i$ and $r_j$, as we saw earlier, we prefix the the name of the relation from which the attribute originally came, before the attribute name.

For example, the relation schema for the Cartesian product of relations *instructor* and *teaches* is:

(*instructor.ID*, *instructor.name*, *instructor.dept_name*, *instructor.salary*
*teaches.ID*, *teaches.course_id*, *teaches.sec_id*, *teaches.semester*, *teaches.year*)

With this schema, we can distinguish *instructor.ID* from *teaches.ID*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema as:

(*instructor.ID*, *name*, *dept_name*, *salary*
*teaches.ID*, *course_id*, *sec_id*, *semester*, *year*)

*instructor*

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 60000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 58583 | Califieri | History | 62000 |
| 76543 | Singh | Finance | 80000 |
| 76766 | Crick | Biology | 72000 |
| 83821 | | | |
| 98345 | | | |

*teaches*

| *ID* | *course_id* | *sec_id* | *semester* | *year* |
|---|---|---|---|---|
| 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | FIN-201 | 1 | Spring | 2010 |
| 15151 | MU-199 | 1 | Spring | 2010 |
| 22222 | PHY-101 | 1 | Fall | 2009 |
| 32343 | HIS-351 | 1 | Spring | 2010 |
| 45565 | CS-101 | 1 | Spring | 2010 |
| 45565 | CS-319 | 1 | Spring | 2010 |
| 76766 | BIO-101 | 1 | Summer | 2009 |
| | | | | 2010 |
| | | | | 2009 |
| | | | | 2009 |
| | | | | 2010 |
| | | | | 2009 |

| *Inst.ID* | *name* | *dept_name* | *salary* | *teaches.ID* | *course_id* | *sec_id* | *semester* | *year* |
|---|---|---|---|---|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 12121 | FIN-201 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 12121 | Wu | Finance | 90000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 12121 | Wu | Finance | 90000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 12121 | Wu | Pinance | 90000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | Wu | Pinance | 90000 | 12121 | FIN-201 | 1 | Spring | 2010 |
| 12121 | Wu | Finance | 90000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 12121 | Wu | Pinance | 90000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

If we only wished to find instructor names and course identifiers for instructors in the Computer Science department, we could add an extra predicate to the **where** clause, as shown below.

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID and instructor.dept_name = 'Comp. Sci.';
```

| *name* | *course_id* |
|---|---|
| Srinivasan | CS-101 |
| Srinivasan | CS-315 |
| Srinivasan | CS-347 |
| Wu | FIN-201 |
| Mozart | MU-199 |
| Einstein | PHY-101 |
| El Said | HIS-351 |
| Katz | CS-101 |
| Katz | CS-319 |
| Crick | BIO-101 |
| Crick | BIO-301 |
| Brandt | CS-190 |
| Brandt | CS-190 |
| Brandt | CS-319 |
| Kim | EE-181 |

## The Natural Join

In our example query that combined information from the instructor and teaches table, the matching condition required instructor.ID to be equal to teaches.ID. These are the only attributes in the two relations that have the same name.

The **natural join** operation operates on two relations and produces a relation as the result. Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations. So, going back to the example of the relations *instructor* and *teaches*, computing *instructor* **natural join** *teaches* considers only those pairs of tuples where both the tuple from *instructor* and the tuple from *teaches* have the same value on the common attribute, *ID*.

Consider the query "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught", which we wrote earlier as:

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;
```

This query can be written more concisely using the natural-join operation in SQL as:

```
select name, course_id
from instructor natural join teaches;
```

Both of the above queries generate the same result.

| ID | name | dept_name | salary | course_id | sec_id | semester | year |
|---|---|---|---|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 | CS-101 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | CS-315 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | CS-347 | 1 | Fall | 2009 |
| 12121 | Wu | Finance | 90000 | FIN-201 | 1 | Spring | 2010 |
| 15151 | Mozart | Music | 40000 | MU-199 | 1 | Spring | 2010 |
| 22222 | Einstein | Physics | 95000 | PHY-101 | 1 | Fall | 2009 |
| 32343 | El Said | History | 60000 | HIS-351 | 1 | Spring | 2010 |
| 45565 | Katz | Comp. Sci. | 75000 | CS-101 | 1 | Spring | 2010 |
| 45565 | Katz | Comp. Sci. | 75000 | CS-319 | 1 | Spring | 2010 |
| 76766 | Crick | Biology | 72000 | BIO-101 | 1 | Summer | 2009 |
| 76766 | Crick | Biology | 72000 | BIO-301 | 1 | Summer | 2010 |
| 83821 | Brandt | Comp. Sci. | 92000 | CS-190 | 1 | Spring | 2009 |
| 83821 | Brandt | Comp. Sci. | 92000 | CS-190 | 2 | Spring | 2009 |
| 83821 | Brandt | Comp. Sci. | 92000 | CS-319 | 2 | Spring | 2010 |
| 98345 | Kim | Elec. Eng. | 80000 | EE-181 | 1 | Spring | 2009 |

The natural join of the *instructor* relation with the *teaches* relation.

For example, suppose we wish to answer the query "List the names of instructors along with the the titles of courses that they teach." The query can be written in SQL as follows:

```
select name, title
from instructor natural join teaches, course
where teaches.course_id= course.course_id;
```

# Additional Basic Operations

### The Rename Operation

Consider again the query that we used earlier:

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;
```

The result of this query is a relation with the following attributes:

*name, course_id*

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.

SQL provides a way of renaming the attributes of a result relation. It uses the **as** clause, taking the form:

*old-name* **as** *new-name*

The **as** clause can appear in both the **select** and **from** clauses.[4]

For example, if we want the attribute name *name* to be replaced with the name *instructor_name*, we can rewrite the preceding query as:

```
select name as instructor_name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;
```

The **as** clause is particularly useful in renaming relations. One reason to rename a relation is to replace a long relation name with a shortened version that is more convenient to use elsewhere in the query. To illustrate, we rewrite the query "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

```
select T.name, S.course_id
from instructor as T, teaches as S
where T.ID= S.ID;
```

Another reason to rename a relation is a case where we wish to compare tuples in the same relation. We then need to take the Cartesian product of a relation with itself and, without renaming, it becomes impossible to distinguish one tuple from the other. Suppose that we want to write the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department." We can write the SQL expression:

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

**String Operations**

Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:

- Percent (%): The % character matches any substring.
- Underscore (_): The _ character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Intro%' matches any string beginning with "Intro".
- '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. to Computer Science', and 'Computational Biology'.
- '_ _ _' matches any string of exactly three characters.
- '_ _ _%' matches any string of at least three characters.

SQL expresses patterns by using the **like** comparison operator. Consider the query "Find the names of all departments whose building name includes the substring 'Watson'." This query can be written as:

```
select dept_name
from department
where building like '%Watson%';
```

For patterns to include the special pattern characters (that is, % and _), SQL allows the specification of an escape character. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a **like** comparison using the **escape** keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:

- **like** 'ab\%cd%' **escape** '\' matches all strings beginning with "ab%cd".
- **like** 'ab\\cd%' **escape** '\' matches all strings beginning with "ab\cd".

## Attribute Specification in Select Clause

The asterisk symbol " * " can be used in the **select** clause to denote "all attributes." Thus, the use of *instructor*.* in the **select** clause of the query:

```
select instructor.*
from instructor, teaches
where instructor.ID= teaches.ID;
```

indicates that all attributes of *instructor* are to be selected. A **select** clause of the form **select** * indicates that all attributes of the result relation of the **from** clause are selected.

## Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

  **select distinct** *name*
  **from** *instructor*
  **order by** *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name*, *name*

## Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, $\geq$ $90,000 and $\leq$ $100,000)

```
select name
from instructor
where salary between 90000 and 100000
```

instead of:

```
select name
from instructor
where salary <= 100000 and salary >= 90000;
```

- Tuple comparison

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID and dept_name = 'Biology';
```

## In Operator

- The **in** operator allows you to specify multiple values in a **where** clause
- The **in** operator is a shorthand for multiple **or** conditions

  **select** *name*
  **from** *instructor*
  **where** *dept_name* **in** ('Comp. Sci.', 'Biology')

## Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010
  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  **union**
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)
- Find courses that ran in Fall 2009 and in Spring 2010
  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  **intersect**
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)
- Find courses that ran in Fall 2009 but not in Spring 2010
  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  **except**
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions **union all**, **intersect all**, and **except all**.
- Suppose a tuple occurs $m$ times in $r$ and $n$ times in $s$, then, it occurs:
  - $m + n$ times in $r$ **union all** $s$
  - $\min(m, n)$ times in $r$ **intersect all** $s$
  - $\max(0, m - n)$ times in $r$ **except all** $s$

## Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values
  - Example: Find all instructors whose salary is null
    **select** *name*
    **from** *instructor*
    **where** *salary* **is null**
- It is not possible to test for **null** values with comparison operators, such as =, <, or <>
  We need to use the **is null** and **is not null** operators instead

- Three values – **true, false, unknown**
- Any comparison with *null* returns *unknown*
  - Example: 5 < *null* or *null* <> *null* or *null* = *null*
- Three-valued logic using the value unknown:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*,
    (*unknown* **and** *unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*
  - "*P* **is unknown**" evaluates to *true* if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

SQL uses the special keyword **null** in a predicate to test for a null value. Thus, to find all instructors who appear in the *instructor* relation with null values for *salary*, we write:

```
select name
from instructor
where salary is null;
```

## Aggregate Functions

*Aggregate functions* are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

### Basic Aggregation

- Find the average salary of instructors in the Computer Science department

```
select avg (salary)
from instructor
where dept_name = 'Comp. Sci';
```

- Find the total number of instructors who teach a course in the Spring 2010 semester

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2010;
```

- Find the number of tuples in the *course* relation

```
select count (*)
from courses;
```

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 76766 | Crick | Biology | 72000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
| 32343 | El Said | History | 60000 |
| 58583 | Califieri | History | 62000 |
| 15151 | Mozart | Music | 40000 |
| 33456 | Gold | Physics | 87000 |
| 22222 | Einstein | Physics | 95000 |

### Aggregation with Grouping

As an illustration, consider the query "Find the average salary in each department." We write this query as follows:

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;
```

| *dept_name* | *avg_salary* |
|---|---|
| Biology | 72000 |
| Comp. Sci. | 77333 |
| Elec. Eng. | 80000 |
| Finance | 85000 |
| History | 61000 |
| Music | 40000 |
| Physics | 91000 |

As another example of aggregation on groups of tuples, consider the query "**Find the number of instructors in each department who teach a course in the Spring 2010 semester.**"

```
select dept_name, count (distinct ID) as instr_count
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name;
```

| *dept_name* | *instr_count* |
|---|---|
| Comp. Sci. | 3 |
| Finance | 1 |
| History | 1 |
| Music | 1 |

## Having Clause

Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg(salary)
from instructor
group by dept_name
having avg(salary) > 42000;
```

| *dept_name* | *avg(avg_salary)* |
|---|---|
| Physics | 91000 |
| Elec. Eng. | 80000 |
| Finance | 85000 |
| Comp. Sci. | 77333 |
| Biology | 72000 |
| History | 61000 |

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

- Total all salaries

```
select sum (salary)
from instructor;
```

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

## Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries
- A **subquery** is a **select-from-where** expression that is nested within another query
- The nesting can be done in the following SQL query

  **select** $A_1, A_2, \ldots, A_n$
  **from** $r_1, r_2, \ldots, r_m$
  **where** $P$

  as follows:
  - $A_i$ can be replaced by a subquery that generates a single value
  - $r_i$ can be replaced by any valid subquery
  - $P$ can be replaced with an expression of the form:

    $B$ <operation> (subquery)

    where $B$ is an attribute and <operation> to be defined later

## Set Membership

SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

- Find courses offered in Fall 2009 and in Spring 2010. (**intersect** example)

```
select distinct course_id
from section
where semester ='Fall' and year = 2009 and
      course_id in (select course_id
                    from section
                    where semester ='Spring' and year = 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010. (**except** example)

```
select distinct course_id
from section
where semester ='Fall' and year = 2009 and
      course_id not in (select  course_id
                        from section
                        where semester ='Spring' and year = 2010);
```

- Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101

```
select count (distinct  ID)
from takes
where (course_id, sec_id, semester, year) in
                    (select course_id, sec_id, semester, year
                    from teaches
                    where teaches.ID = 10101);
```

- Note: Above query can be written in simpler manner. The formulation above is simply to illustrate SQL features.

## Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query "**Find the names of all instructors whose salary is greater than at least one instructor in the Biology department**."

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > **some.** This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

Some

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Biology');
```

SQL also allows < **some**, <= **some**, >= **some**, = **some**, and <> **some** comparisons. As an exercise, verify that = **some** is identical to **in**, whereas <> **some** is *not* the same as **not in**.[8]

Let us **find the names of all instructors that have a salary value greater than that of each instructor in the Biology department.** The construct **> all** corresponds to the phrase "**greater than all.**" Using this construct, we write the query as follows:

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = 'Biology');
```

As it does for **some**, SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons. As an exercise, verify that <> **all** is identical to **not in**, whereas = **all** is *not* the same as **in**.

- The **exists** construct returns the value **true** if the argument subquery is nonempty
  - **exists** $r \Leftrightarrow r \neq \emptyset$
  - **not exists** $r \Leftrightarrow r = \emptyset$

- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

```
select course_id
from section as S
where semester = 'Fall' and year = 2009 and
      exists (select *
              from section as T
              where semester = 'Spring' and year = 2010
              and S.course_id = T.course_id);
```

- **Correlation name** – variable $S$ in the outer query
- **Correlated subquery** – the inner query

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                   (select T.course_id
                    from takes as T
                    where S.ID = T.ID));
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took
- Note: $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = **all** and its variants

- The **unique** construct tests whether a subquery has any duplicate tuples in its result
- The **unique** construct evaluates to "true" if a given subquery contains no duplicates
- Find all courses that were offered at most once in 2009

```
select  T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id = R.course_id
              and R.year = 2009);
```

## Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000

```
select dept_name, avg_salary
from (select dept_name, avg(salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name) as  dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

## The With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs
- Find all departments with the maximum budget

```
with max_budget(value) as
      (select max(budget)
      from department)
select department.name
from department, max_budget
where department.budget=max_budget.value;
```

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as
                select dept_name, sum(salary)
                from instructor
                group by dept_name,
        dept_total_avg(value) as
                (select avg(value)
                from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

Complex query for with statement.

## Modifications of the Database

### Deletion

- Delete all instructors

```
delete from instructor
```

- Delete all instructors from the Finance department

```
delete from instructor
where dept_name= 'Finance';
```

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson');
```

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
  a) First, compute **avg** (salary) and find all tuples to delete
  b) Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

## Insertion

- Add a new tuple to course

  **insert into** *course*
  **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently:

  **insert into** *course (course_id, title, dept_name, credits)*
  **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to student with *tot_creds* set to null

  **insert into** *student*
  **values** ('3003', 'Green', 'Finance', *null*);

- Add all instructors to the *student* relation with *tot_creds* set to 0

  **insert into** *student*
  **select** *ID, name, dept_name*, 0
  **from** *instructor*

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation
- Otherwise queries like

  **insert into** *table1* **select * from** *table1*

  would cause problem

## Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

    **update** *instructor*
    **set** $salary = salary * 1.03$
    **where** $salary > 100000$;
    **update** *instructor*
    **set** $salary = salary * 1.05$
    **where** $salary <= 100000$;

- The order is important
- Can be done better using the **case** statement
- Same query as before but with **case** statement

```
update instructor
set salary = case
                when salary <= 100000
                then salary * 1.05
                else salary * 1.03
                end
```

- Recompute and update tot_creds value for all students

```
update student S
set tot_creds = (select sum(credits)
                 from takes, course
                 where takes.course_id = course.course_id and
                 S.ID = takes.ID and
                 takes.grade <> 'F' and
                 takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
when sum(credits) is not null then sum(credits)
else 0
end
```