國立陽明交通大學

NATIONAL YANG MING CHIAO TUNG UNIVERSITY

# Biological Databases: Theories and Practice

430032

## Introduction to SQL

**Instructor: Prof. LEE, Tzong-Yi (李宗夷)**
**Email:** leetzongyi@nycu.edu.ctw

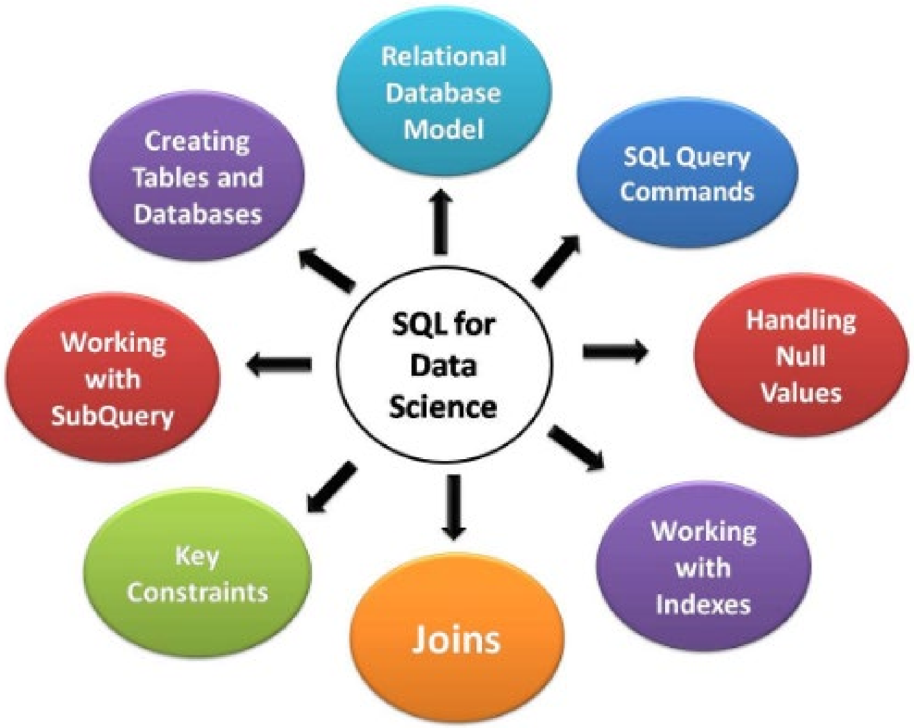
***Professor***
***Institute of Bioinformatics and Systems Biology,***
***National Yang Ming Chiao Tung University***

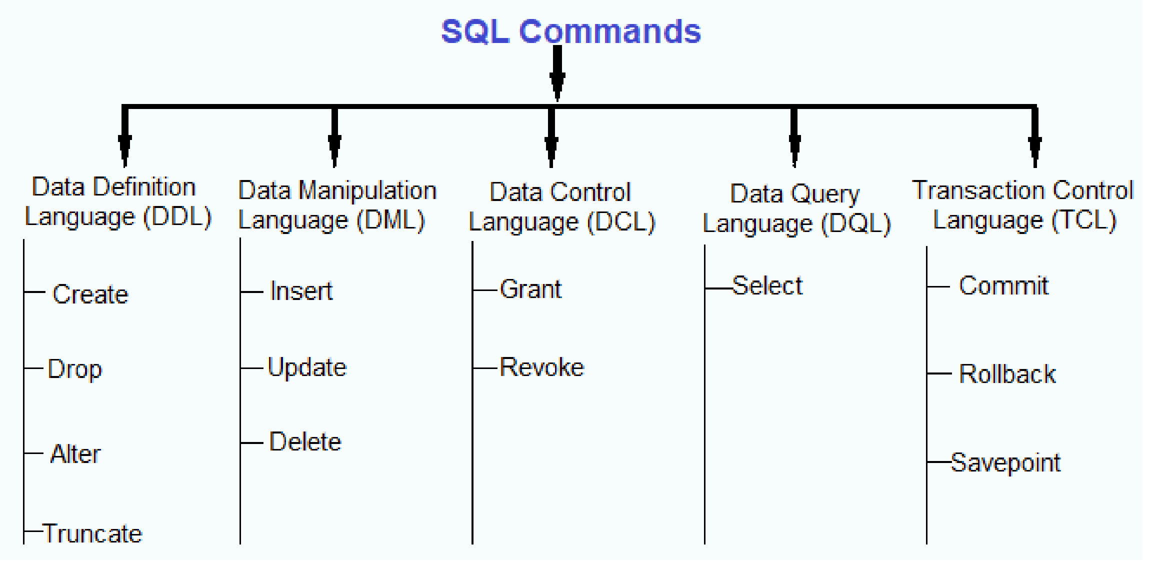
# Outline

- Overview of The SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

# What is SQL?

- **SQL (Structured Query Language**) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e., data incorporating relations among entities and variables.



# How to get started?

- How to construct the table **instructor** in database?
- How to input instances into the constructed table?
- How to search specific instances against the table?

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

https://www.w3schools.com/mysql/mysql_create_table.asp





# MySQL CREATE TABLE Statement

❮ Previous

## The MySQL CREATE TABLE Statement

The `CREATE TABLE` statement is used to create a new table in a database.

### Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

**Tip:** For an overview of the available data types, go to our complete Data Types Reference.

## MySQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

# Domain Types Commonly Used in SQL

- **char(n):** Fixed length character string, with user-specified length *n.*
- **varchar(n):** Variable length character strings, with user-specified maximum length *n.*
- **int:** Integer (a finite subset of the integers that is machine-dependent).
- **smallint:** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d):** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point.
- **float(n):** Floating point number, with user-specified precision of at least *n* digits.
- **double:** Floating point and double-precision floating point numbers, with machine-dependent precision.

# MySQL Data Types

| DATE TYPE | SPEC | DATA TYPE | SPEC |
|---|---|---|---|
| CHAR | String (0 - 255) | INT | Integer (-2147483648 to 214748-3647) |
| VARCHAR | String (0 - 255) | BIGINT | Integer (-9223372036854775808 to 9223372036854775807) |
| TINYTEXT | String (0 - 255) | FLOAT | Decimal (precise to 23 digits) |
| TEXT | String (0 - 65535) | DOUBLE | Decimal (24 to 53 digits) |
| BLOB | String (0 - 65535) | DECIMAL | "DOUBLE" stored as string |
| MEDIUMTEXT | String (0 - 16777215) | DATE | YYYY-MM-DD |
| MEDIUMBLOB | String (0 - 16777215) | DATETIME | YYYY-MM-DD HH:MM:SS |
| LONGTEXT | String (0 - 4294967295) | TIMESTAMP | YYYYMMDDHHMMSS |
| LONGBLOB | String (0 - 4294967295) | TIME | HH:MM:SS |
| TINYINT | Integer (-128 to 127) | ENUM | One of preset options |
| SMALLINT | Integer (-32768 to 32767) | SET | Selection of preset options |
| MEDIUMINT | Integer (-8388608 to 8388607) | BOOLEAN | TINYINT(1) |

# Create Table and Insert Data

- An SQL relation is defined using the **create table** command:

  **create table** $r$ ($A_1$ $D_1$, $A_2$ $D_2$, ..., $A_n$ $D_n$,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - $r$ is the name of the relation
  - each $A_i$ is an attribute name in the schema of relation $r$
  - $D_i$ is the data type of values in the domain of attribute $A_i$

- Example:

  **create table** *instructor* (
  *ID* **char**(5),
  *name* **varchar**(20) **not null,**
  *dept_name* **varchar**(20),
  *salary* **int**(10))

- **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
- **insert into** *instructor* **values** ('10211', null, 'Biology', 66000);

# Table description

- Show the table structure by using *describe* command:

```
MariaDB [student]> describe instructor;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| ID        | char(5)     | YES  |     | NULL    |       |
| name      | varchar(20) | NO   |     | NULL    |       |
| dept_name | varchar(20) | YES  |     | NULL    |       |
| salary    | int(10)     | YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

# Integrity Constraints in Create Table

- **not null**
- **primary key** ($A_1$, ..., $A_n$ )
- **foreign key** ($A_m$, ..., $A_n$ ) **references** *r*

Example: Declare ***ID*** as the primary key for ***instructor***

**create table** *instructor* (
*ID* **char**(5),
*name* **varchar**(20) **not null,**
*dept_name* **varchar**(20),
*salary* **int**(10),
**primary key** (*ID*),
**foreign key** *(dept_name*) **references** *department (dept_name) );*

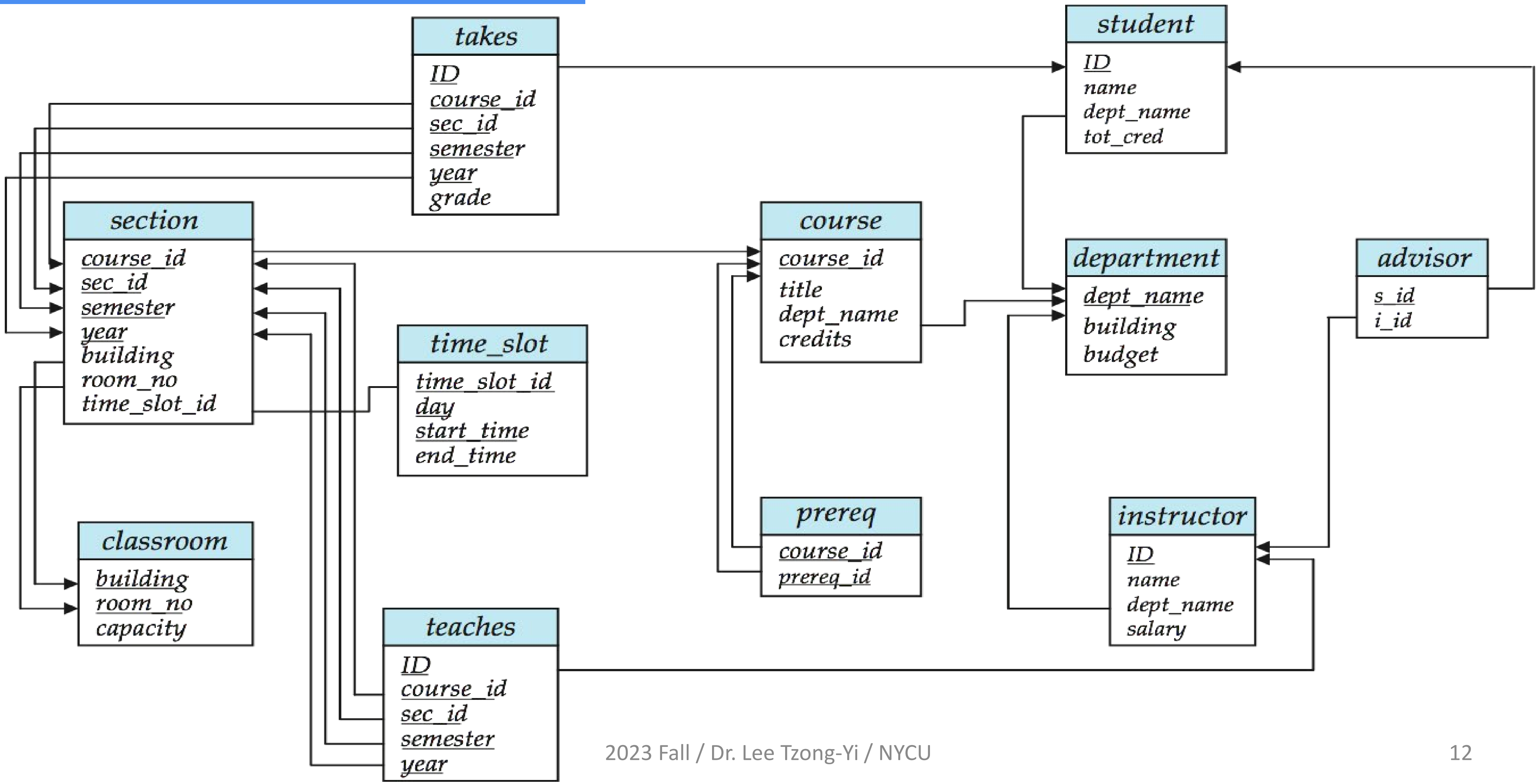
**primary key** declaration on an attribute automatically ensures **uniqueness** and **not null**

# And a Few More Relation Definitions

- **create table** *student* (
  *ID* **varchar**(5) **primary key**,
  *name* **varchar**(20) not null,
  *dept_name* **varchar**(20),
  *tot_cred* **int**(3),
  **foreign key** (*dept_name*) **references** *department* (*dept_name*) );

- **create table** *takes* (
  *ID* **varchar**(5) **primary key**,
  *course_id* **varchar**(8),
  *sec_id* **varchar**(8),
  *semester* **varchar**(6),
  *year* **int**(4),
  *grade* **varchar**(2),
  **foreign key** (*ID*) **references** *student,*
  **foreign key** (*course_id, sec_id, semester, year*)
  **references** *section* (*course_id, sec_id, semester, year*)
  );

# Create Tables according to Schema Diagram



# Drop and Alter Table Attributes

- **drop table**
- **alter table**
  - **alter table** *r* **add** *A D*
    - where ***A*** is the name of the attribute to be added to relation ***r*** and ***D*** is the domain of ***A.***
    - All tuples in the relation are assigned ***null*** as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where ***A*** is the name of an attribute of relation ***r***
    - Dropping of attributes not supported by any tables.

# Data Insertion

- Add a new tuple to *course*

  **insert into** *instructor*
  **values** ('A0001', 'Tomas Huang', 'Computer Science', 80000);

- or equivalently

  **insert into** *instructor* (*ID*, name, *dept_name*, salary)
  **values** ('A0001', 'Tomas Huang', 'Computer Science', 80000);

- Add a new tuple to *instructor* with *salary* set to **null**

  **insert into** *instructor*
  **values (**'A0002', 'Alex Chen', 'Biology', *null*);

# Basic Query Structure

- A typical SQL query has the form:

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

  - $\boldsymbol{A_i}$ represents an attribute
  - $\boldsymbol{r_i}$ represents a relation
  - $\boldsymbol{P}$ is a predicate.

- The result of an SQL query is a relation.

# The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  **select** *name*

  **from** *instructor*

- NOTE: **SQL names are case insensitive** (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

# The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the names of all departments with instructor, and remove duplicates

  **select distinct** *dept_name*
  **from** *instructor*

- The keyword **all** specifies that duplicates not be removed.

  **select all** *dept_name*
  **from** *instructor*

# The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

  **select** *
  **from** *instructor*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
- The query:

  **select** *ID, name, salary/12*
  **from** *instructor*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

# The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept with salary > 80000

  **select** *name*
  **from** *instructor*
  **where** *dept_name* = 'Comp. Sci.' **and** *salary* > 80000
- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

# The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  **select** □
  **from** *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations.
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

# Cartesian Product

| *Inst.ID* | *name* | *dept_name* | *salary* | *teaches.ID* | *course_id* | *sec_id* | *semester* | *year* |
|---|---|---|---|---|---|---|---|---|
| 10101 | Srinivasan | Physics | 95000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Physics | 95000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Physics | 95000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Physics | 95000 | 10101 | FIN-201 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Physics | 95000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Physics | 95000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 12121 | Wu | Physics | 95000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 12121 | Wu | Physics | 95000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 12121 | Wu | Physics | 95000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | Wu | Physics | 95000 | 10101 | FIN-201 | 1 | Spring | 2010 |
| 12121 | Wu | Physics | 95000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 12121 | Wu | Physics | 95000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | .. | ... | ... | ... | ... |

# Joins

- For all instructors who have taught courses, find their names and the course ID of the courses they taught.

**select** *name, course_id*
**from** *instructor, teaches*
**where** *instructor.ID = teaches.ID*

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department

**select** *section.course_id, semester, year, title*
**from** *section, course*
**where** *section.course_id = course.course_id* **and**
*dept_name* = 'Comp. Sci.'

# Natural Join

- Natural join matches tuples with the same values for **all common attributes**, and retains only one copy of each common column
  - **select** * **from** *instructor* **natural join** *teaches*;

*instructor*

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einst | Physics | 95000 |
| 32343 | El Sa | | |

*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 |
| | | | Spring | 2010 |
| | | | Fall | 2009 |
| | | | Spring | 2010 |
| | | | Spring | 2010 |
| | | | Spring | 2010 |
| | | | Summer | 2009 |
| | | | Summer | 2010 |
| | | | Spring | 2009 |
| | | | Spring | 2009 |
| | | | Spring | 2010 |
| | | | Spring | 2009 |

| *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 |

# The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

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

- E.g.,
  - **select** *ID, name, salary/12* **as** *monthly_salary* **from** *instructor*

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  - **select distinct** *T. name*

    **from** *instructor* **as** *T, instructor* **as** *S*

    **where** *T.salary > S.salary* **and** *S.dept_name = 'Comp. Sci.'*

- Keyword **as** is optional and may be omitted

  *instructor* **as** *T ≡ instructor T*

# String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using two special characters:
  - percent (%). The **%** character matches **any substring**.
  - underscore (_). The **_** character matches **any character**.
- Find the instructors whose name includes the substring “dar”.

  **select** *name* **from** *instructor* **where** *name* **like** '%dar%'
- Match the string “100 %”

  **like ‘**100 \%' escape character '\'
- SQL supports a variety of string operations such as
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.
  - Reference: https://dev.mysql.com/doc/refman/8.0/en/string-functions.html#function_substring

# 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 by 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, ≥ \$90,000 and ≤ \$100,000)
  - **select** *name*
    **from** *instructor*
    **where** *salary* **between** 90000 **and** 100000
- Tuple comparison
  - **select** *name*, *course_id*
    **from** *instructor*, *teaches*
    **where** (*instructor*.*ID*, *dept_name*) = (*teaches*.*ID*, ’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

- 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**

# Null Values and Three Valued Logic

- **Any comparison with *null* returns *unknown***
  - Example*: 5 < null   or   null <> null    or    null = null*
- Three-valued logic using the truth 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*

# Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

**avg:** average value
**min:** minimum value
**max:** maximum value
**sum:** sum of values
**count:** number of values

# Aggregate Functions (Cont.)

- 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** *course*;

# Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **from** *instructor* **group by** *dept_name*;

| *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 |

average salary

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

# Aggregate Functions – 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;

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

# Null Values and Aggregates

- 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.

- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

# Example Query

- Find courses offered in Fall 2009 and in Spring 2010

**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

**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);

# Example Query

- 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 a much simpler manner. The formulation above is simply to illustrate SQL features.

# Set Comparison

- Find names of instructors with salary greater than that of some (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';

- Same query using > **some** clause

**select** *name*
**from** *instructor*
**where** *salary* > **some** (**select** *salary*
**from** *instructor*
**where** *dept name* = 'Biology');

# Definition of Some Clause

- F <comp> **some** *r* $\Leftrightarrow \exists\, t \in r$ such that (F <comp> *t* )
  Where <comp> can be: $<, \le, >, =, \neq$

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$ ) = true

(read: 5 < some tuple in the relation)

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$ ) = false

(5 = **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$ ) = true

(5 $\neq$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$ ) = true (since 0 $\neq$ 5)

(= **some**) $\equiv$ **in**
However, ($\neq$ **some**) $\not\equiv$ **not in**

# Example Query

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

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

# Definition of all Clause

- F <comp> **all** *r* $\Leftrightarrow \forall\, t \in r$ (F <comp> *t)*

(5 < **all** $\begin{array}{|c|}\hline 0\\ \hline 5\\ \hline 6\\ \hline\end{array}$ ) = false

(5 < **all** $\begin{array}{|c|}\hline 6\\ \hline 10\\ \hline\end{array}$ ) = true

(5 = **all** $\begin{array}{|c|}\hline 4\\ \hline 5\\ \hline\end{array}$ ) = false

(5 $\neq$ **all** $\begin{array}{|c|}\hline 4\\ \hline 6\\ \hline\end{array}$ ) = true (since 5 $\neq$ 4 and 5 $\neq$ 6)

($\neq$ **all**) $\equiv$ **not in**
However, (= **all**) $\not\equiv$ **in**

# Test for Empty Relations

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

# Correlation Variables

- 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*);

- **Correlated subquery**
- **Correlation name** or **correlation variable**

# Not Exists

- 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));
```

- Note that $X - Y = \emptyset \iff X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

# Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- 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);
```

# Derived Relations

- 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;

# Scalar Subquery

```
select dept_name,
        (select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

# Modification 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');

# Example Query

- 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:
  1. First, compute **avg** salary and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

# Modification of the Database – 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*);

# Modification of the Database – Insertion

- 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** *table*2 **select * from** *table*1
  would cause problems)

# Modification of the Database – Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise
  - 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 (next slide)

# Case Statement for Conditional Updates

- 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**

# Thank you and any questions?