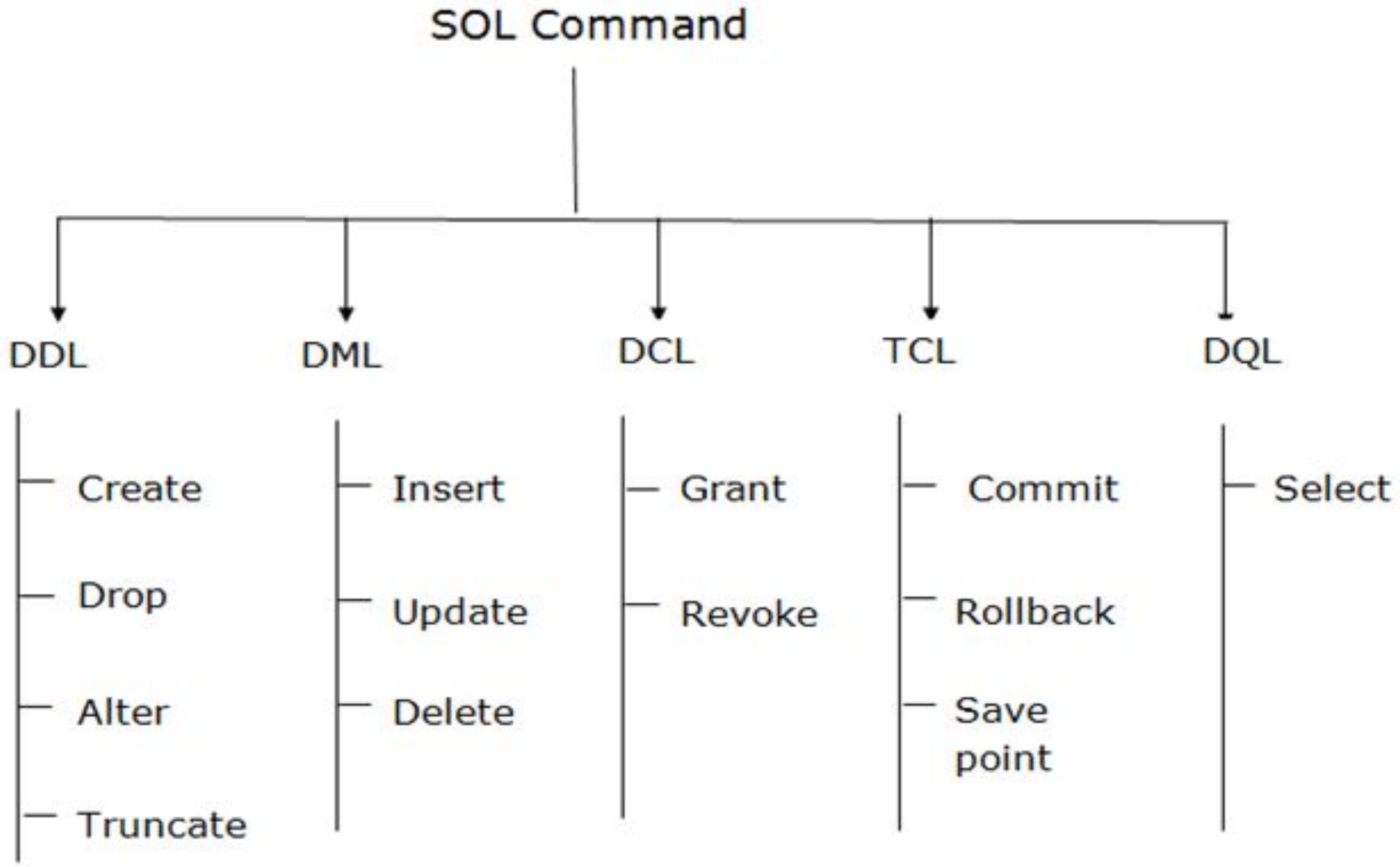
SQL

Unit III

SQL

- SQL (Structured Query Language) is used to perform operations on the records stored in the database, such as updating records, inserting records, deleting records, creating and modifying database tables, views, etc.
- SQL is not a database system, but it is a query language.
- This database language is mainly designed for maintaining the data in relational database management systems

Types of SQL Commands



DDL

- Data Definition Language (DDL)
  - DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
  - 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
  - All the command of DDL are auto-committed that means it permanently save all the changes in the database.
  - Here are some commands that come under DDL:
  - **CREATE**
  - **ALTER**
  - **DROP**
  - **TRUNCATE**

Basic DDL commands

- CREATE - to create a database and its objects like (table, index, views, store procedure, function, and triggers)
- ALTER - alters the structure of the existing database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- RENAME - rename an object

Domain Types in SQL

- 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 nor 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.

Create Table Construct

- 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)**,**
  dept_name **varchar**(20),
  salary **numeric**(8,2))

Integrity Constraints in Create Table

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

Example:

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

primary key primary key (A_{j1}, A_{j2},..., A_{jm}): The primary-key specification says that attributes A_{j1}, A_{j2},..., A_{jm} form the primary key for the relation.

- foreign key (A_{j1}, A_{j2},…, A_{jm}) references ***s***:
  - The foreign key specification says that the values of attributes (A_{j1}, A_{j2},…, A_{jm}) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation ***s***.

And a Few More Relation Definitions

- **create table** *student* (
```
        ID              varchar(5),
        name            varchar(20) not null,
        dept_name       varchar(20),
        tot_cred        numeric(3,0),
        primary key (ID),
        foreign key (dept_name) references department);
```

- **create table** *takes* (
```
        ID              varchar(5),
        course_id       varchar(8),
        sec_id          varchar(8),
        semester        varchar(6),
        year            numeric(4,0),
        grade           varchar(2),
        primary key (ID, course_id, sec_id, semester, year) ,
        foreign key (ID) references  student,
        foreign key (course_id, sec_id, semester, year) references
section);
```
  - Note: *sec_id* can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

And more still

```
• create table course (
        course_id       varchar(8),
        title           varchar(50),
        dept_name        varchar(20),
        credits         numeric(2,0),
         primary key (course_id),
         foreign key (dept_name) references
  department);
```

Updates to tables

- **Insert**
  - **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **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 exiting 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 many databases.

- Relation 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	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- Relation Course

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Basic Query Structure

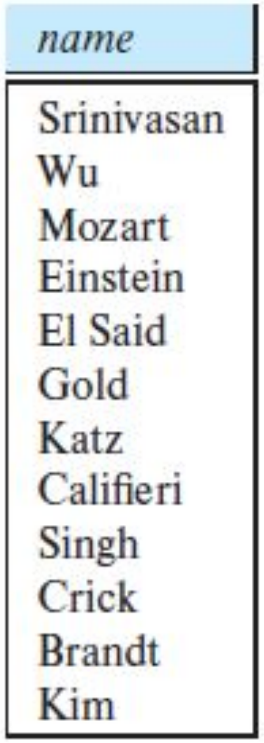
- The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**.
- A query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result
- A typical SQL query has the form:

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

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.

- The result is a relation consisting of a single attribute with the heading ***name***.

- If the relation is *instructor,* then the relation that results from the preceding query is:

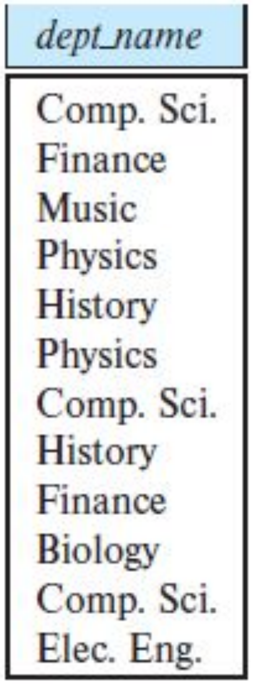


name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

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

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

- Since more than one instructor can belong to a department, a department name could appear more than once in the *instructor* relation



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

- In those cases where we want to force the elimination of duplicates, we insert the keyword distinct after select.

  select distinct *dept name* **from** *instructor***;**

- The keyword **all** specifies that duplicates should not be 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*;

- An asterisk in the select clause denotes "all attributes"

  select *
  from *instructor*

- An attribute can be a literal with no **from** clause

  select '437'

  - Results is a table with one column and a single row with value "437"
  - Can give the column a name using:

    select '437' **as** *FOO*

- An attribute can be a literal with **from** clause

  select 'A'
  from *instructor*

  - Result is a table with one column and *N* rows (number of tuples in the *instructors* table), each row with value "A"

- 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.
  - Can rename "*salary/12*" using the **as** clause:

    select *ID, name, salary/12* **as** *monthly_salary*

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

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.'
- Comparison results can be combined using the logical connectives **and, or,** and **not**
  - To find all instructors in Comp. Sci. dept with salary > 80000

    select *name*
    from *instructor*
    where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000
- Comparisons can be applied to results of arithmetic expressions.

String Operations

- SQL specifies strings by enclosing them in single quotes
- The SQL standard specifies that the equality operation on strings is case sensitive; as a result, the expression “'comp. sci.' = 'Comp. Sci.'” evaluates to false
- SQL also permits a variety of functions on character strings:
  - such as concatenating (using “ || ”),
  - extracting substrings,
  - finding the length of strings,
  - converting strings to uppercase (using the function upper(*s*) where *s* is a string) and lowercase (using the function lower(*s*)),
  - removing spaces at the end of the string (using trim(*s*)), and so on.

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 names of all instructors whose name includes the substring “dar”.

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

  like ‘100 \%' **escape** '\'

  in that above we use backslash (\) as the escape character.

String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
  - ‘Intro%’ matches any string beginning with “Intro”.
  - ‘%Comp%’ matches any string containing “Comp” as a substring.
  - ‘_ _ _’ matches any string of exactly three characters.
  - ‘_ _ _ %’ matches any string of at least three characters.

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- SQL offers the user some control over the order in which tuples in a relation are displayed.
- The order by clause causes the tuples in the result of a query to appear in sorted order.
- 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

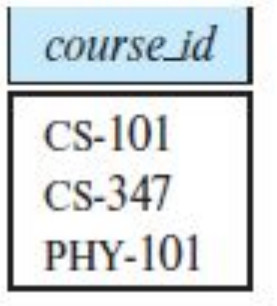
- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Similarly, we can use the not between comparison operator

Set Operations

- The SQL operations union, intersect, and except operate on relations and correspond to the mathematical set operations
- The set of all courses taught in the Fall 2017 semester:

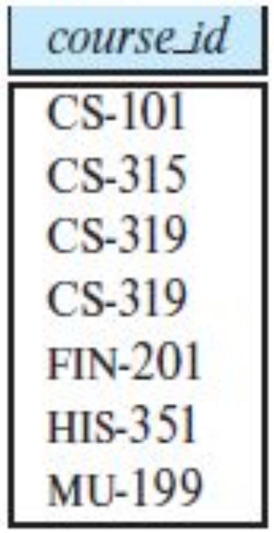
  select *course id* from *section*

  where *semester* = 'Fall' and *year*= 2017;



course_id
CS-101
CS-347
PHY-101

- The set of all courses taught in the Spring 2018 semester:
- select *course id* from *section* where
- *semester* = 'Spring' and *year*= 2018



course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

The Union Operation

- To find the set of all courses taught either in Fall 2017 or in Spring 2018, or both, we write the following query:

(select *course id*

from *section* where

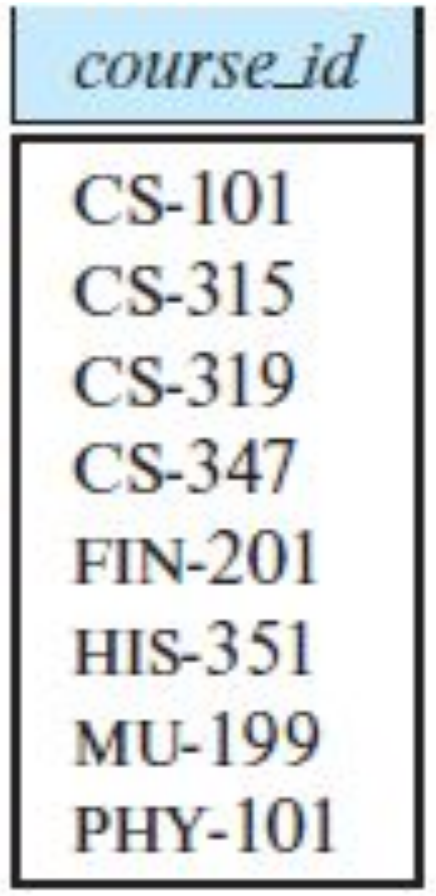
semester = 'Fall' and *year*= 2017)

union

(select *course id*

from *section* where

semester = 'Spring' and *year*= 2018);



course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

The Intersect Operation

- To find the set of all courses taught in both the Fall 2017 and Spring 2018, we write:

(select *course id*

from *section*

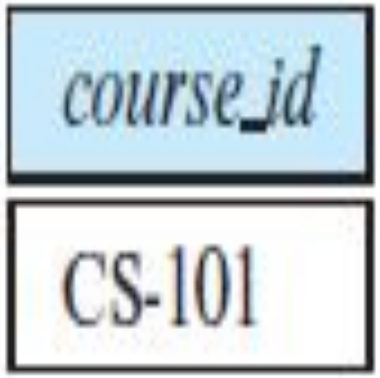
where *semester* = 'Fall' and *year*= 2017)

intersect

(select *course id*

from *section*

where *semester* = 'Spring' and *year*= 2018);



course_id
CS-101

The Except Operation

- To find all courses taught in the Fall 2017 semester but not in the Spring 2018 semester, we write:

```
(select course id
from section
where semester = 'Fall' and year= 2017)
except
(select course id
from section
where semester = 'Spring' and year= 2018);
```

- The except operation, outputs all tuples from its first input that do not occur in the second input; that is, it performs set difference.

Null Values

- Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.
- predicate in a where clause can involve Boolean operations such as and, or, and not on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.
  - and: The result of true and unknown is unknown, false and unknown is false, while unknown and unknown is unknown.
  - or: The result of true or unknown is true, false or unknown is unknown, while unknown or unknown is unknown.
  - not: The result of not unknown is unknown.

- If the where clause predicate evaluates to either false or unknown for a tuple, that tuple is not added to the result.
- 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 standard built-in aggregate functions
- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

Basic Aggregate Functions

- "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 2018 semester."

select count (distinct ID) **from** teaches **where** semester = 'Spring' and year = 2018;

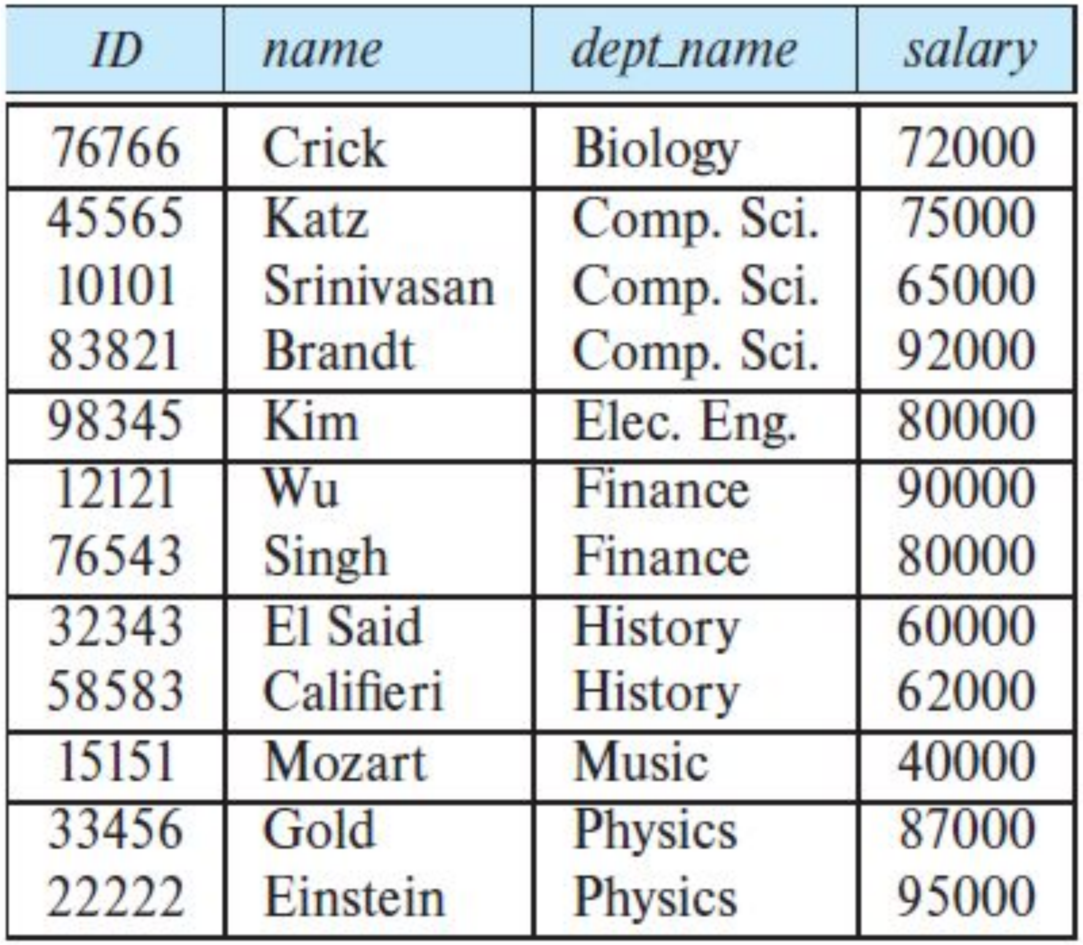
Because of the keyword distinct preceding ID, even if an instructor teaches more than one course, she is counted only once in the result.

- "Find the total number of tuples in the course relation."

select count (*) **from** course;

Aggregation with Grouping

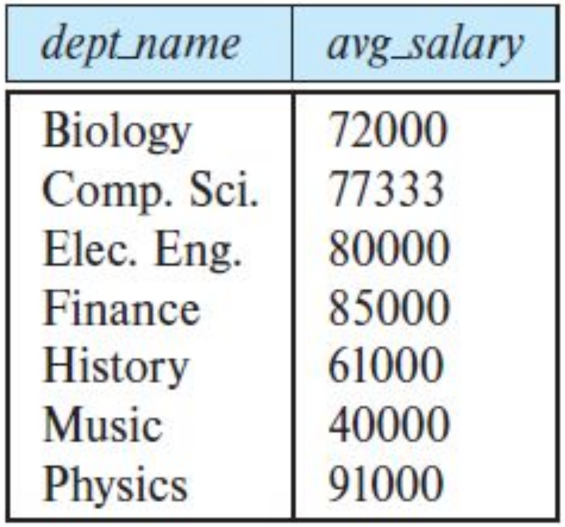
- Tuples with the same value on all attributes in the group by clause are placed in one group
- (Sample of first step: group formation)



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

- “Find the average salary in each department.”

select dept name, **avg** (salary) as avg salary **from** instructor **group by** dept name;

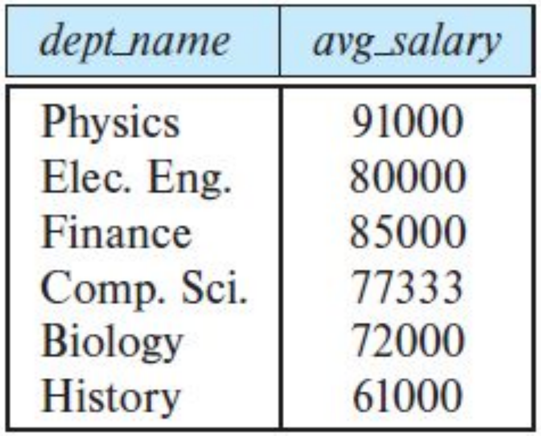


dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

The Having Clause

- At times, it is useful to state a condition that applies to groups rather than to tuples
- For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000
- This condition does not apply to a single tuple; rather, it applies to each **group** constructed by the **group by** clause
- To express such a query, we use the **having** clause of SQL

```
select dept name, avg (salary) as avg sa
 from instructor
group by dept name
having avg (salary) > 42000;
```



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

- The meaning of a query containing aggregation, group by, or having clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the from clause is first evaluated to get a relation.
2. If a where clause is present, the predicate in the where clause is applied on the result relation of the from clause.
3. Tuples satisfying the where predicate are then placed into groups by the group by clause if it is present. If the group by clause is absent, the entire set of tuples satisfying the where predicate is treated as being in one group.

4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the having clause predicate are removed.
5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group
6. E.g. “For each course section offered in 2017, find the average total credits (tot cred) of all students enrolled in the section, if the section has at least 2 students.”
   – **select** course id, semester, year, sec id, avg (tot cred)
   – **from** student, takes
   – **where** student.ID= takes.ID and year = 2017
   – **group by** course id, semester, year, sec id
   – **having** count (ID) >= 2;

Queries on Multiple Relations

- Queries often need to access information from multiple relations.
- “Retrieve the names of all instructors, along with their department names and department building name.”

```
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);

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);
```

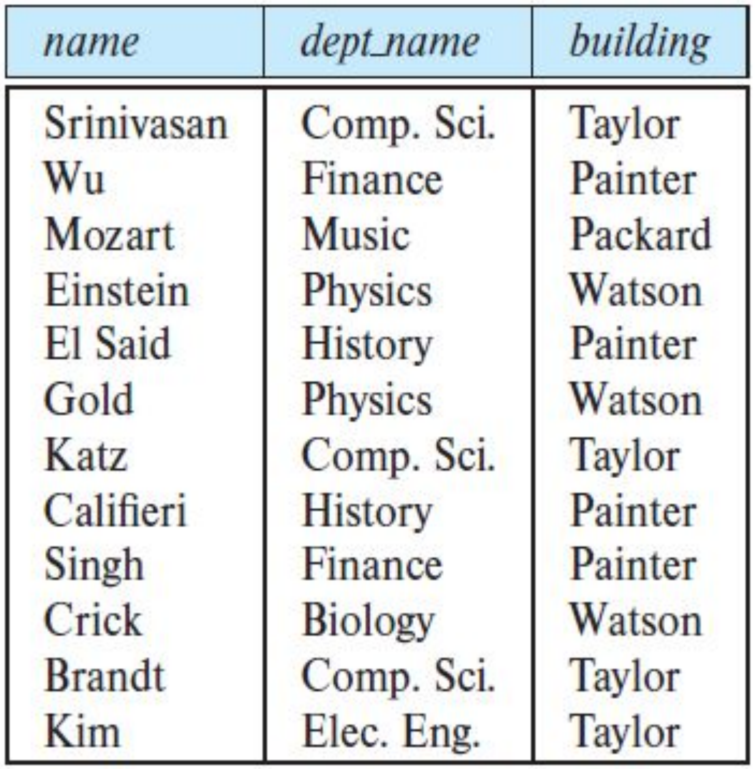
```
create table department
   (dept_name       varchar (20),
    building        varchar (15),
    budget          numeric (12,2),
    primary key (dept_name));

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);

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);
```

- 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.

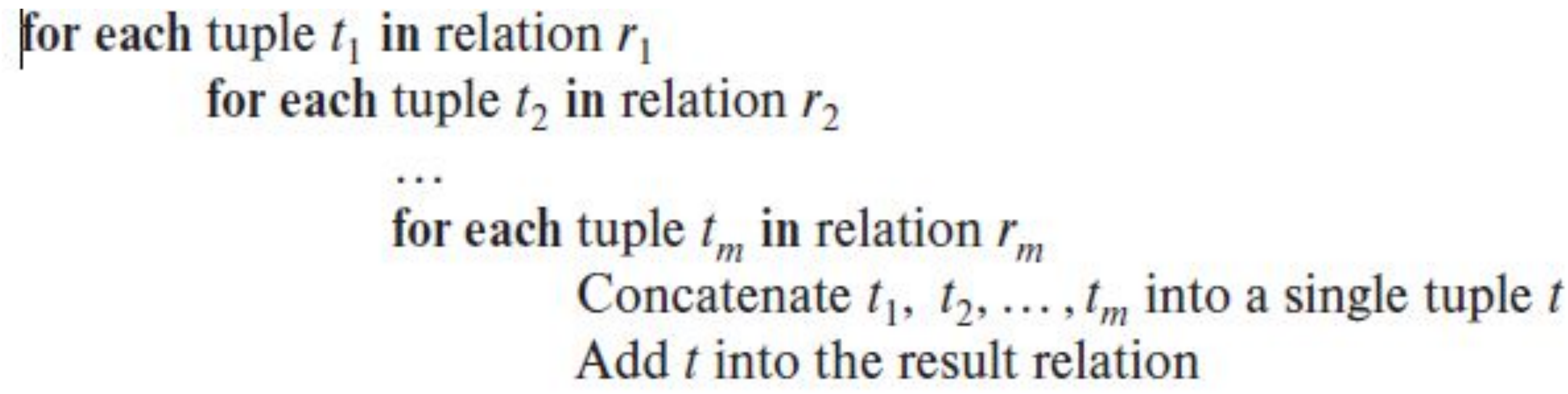
select *name*, *instructor*.*dept name*,
building **from** *instructor*, *department*
where *instructor*.*dept name*=
department.*dept name*;



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

General architecture of the query involving multiple tables

- The from clause by itself defines a Cartesian product of the relations listed in the clause.



```
for each tuple $t_1$ in relation $r_1$
        for each tuple $t_2$ in relation $r_2$
                ...
                for each tuple $t_m$ in relation $r_m$
                        Concatenate $t_1$, $t_2$, ..., $t_m$ into a single tuple $t$
                        Add $t$ into the result relation
```

- same attribute name may appear in both ri and rj , as we saw earlier, we prefix 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)

- The Cartesian product by itself combines tuples from **instructor** and **teaches** that are unrelated to each other.
- Each tuple in **instructor** is combined with every tuple in **teaches**, even those that refer to a different instructor.
- The result can be an extremely large relation, and it rarely makes sense to create such a Cartesian product.

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

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

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...

- Instead, the predicate in the where clause is used to restrict the combinations created by the Cartesian product to those that are meaningful for the desired answer
- We would likely want a query involving instructor and teaches to combine a particular tuple t in instructor with only those tuples in teaches that refer to the same instructor to which t refers.
- That is, we wish only to match teaches tuples with instructor tuples that have the same ID value.

select name, course id

from instructor, teaches

where instructor.ID= teaches.ID;

- Note that the preceding query outputs only instructors who have taught some course.
- Instructors who have not taught any course are not output

- If we wished to find only 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.';

- In general, the meaning of an SQL query can be understood as follows:

1. Generate a Cartesian product of the relations listed in the from clause.
2. Apply the predicates specified in the where clause on the result of Step 1.
3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the select clause.

Self Join Example

- Relation *emp-super*

person	*supervisor*
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of “Bob”
- Find the supervisor of the supervisor of “Bob”
- Find ALL the supervisors (direct and indirect) of “Bob

Joins

- Different types of JOINs in MySQL

  1. MySQL INNER JOIN clause

  2. MySQL OUTER JOINs

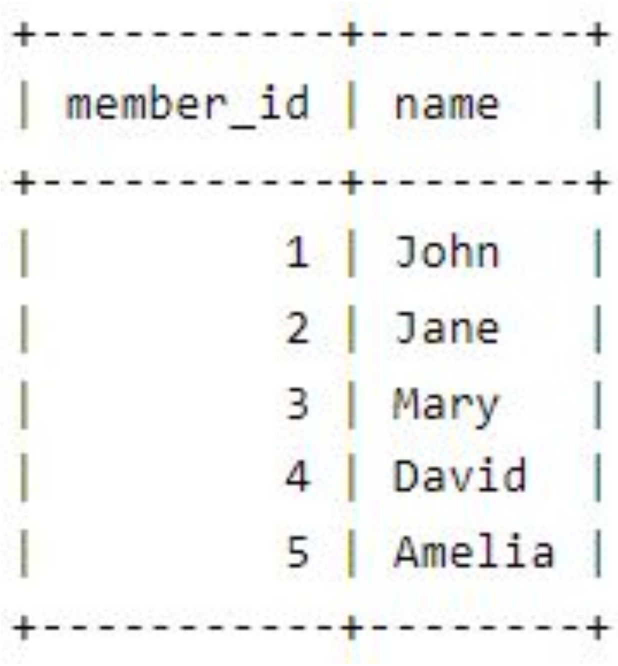
  2.1 MySQL LEFT JOIN clause

  2.2 MySQL RIGHT JOIN clause

  3. MySQL CROSS JOIN clause

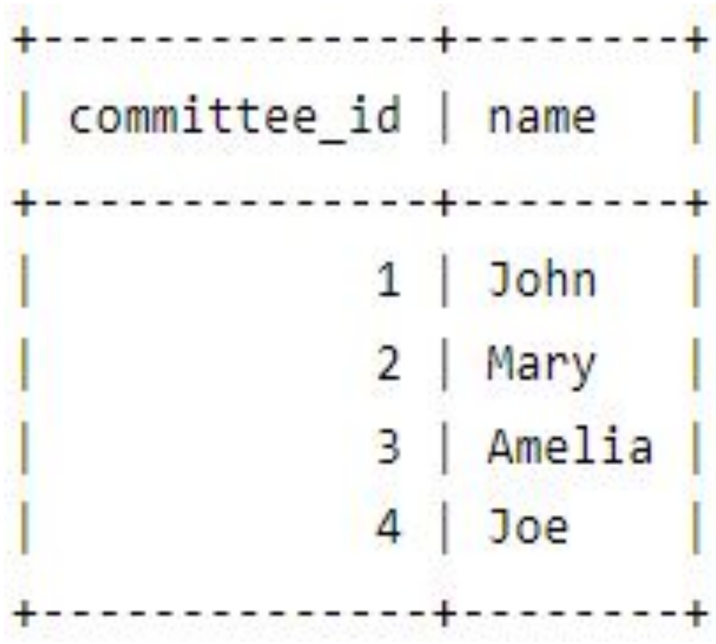
- Consider the tables:

members



```
+-----------+--------+
| member_id | name   |
+-----------+--------+
|         1 | John   |
|         2 | Jane   |
|         3 | Mary   |
|         4 | David  |
|         5 | Amelia |
+-----------+--------+
```

committee

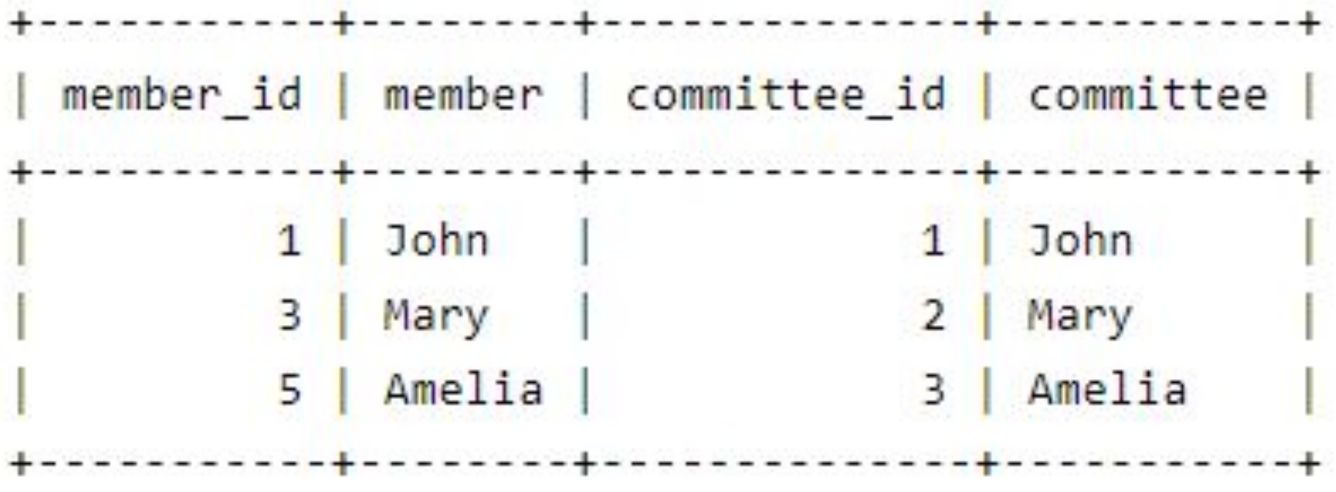


```
+--------------+--------+
| committee_id | name   |
+--------------+--------+
|            1 | John   |
|            2 | Mary   |
|            3 | Amelia |
|            4 | Joe    |
+--------------+--------+
```

- The inner join clause joins two tables based on a condition which is known as a join predicate.
- The inner join clause compares each row from the first table with every row from the second table.
- The inner join clause includes only matching rows from both tables.

- The following statement uses an inner join clause to find members who are also the committee members:

```
SELECT
   m.member_id, m.name AS member,  c.committee_id,
   c.name AS committee
FROM members m
INNER JOIN committees c ON c.name = m.name;
```

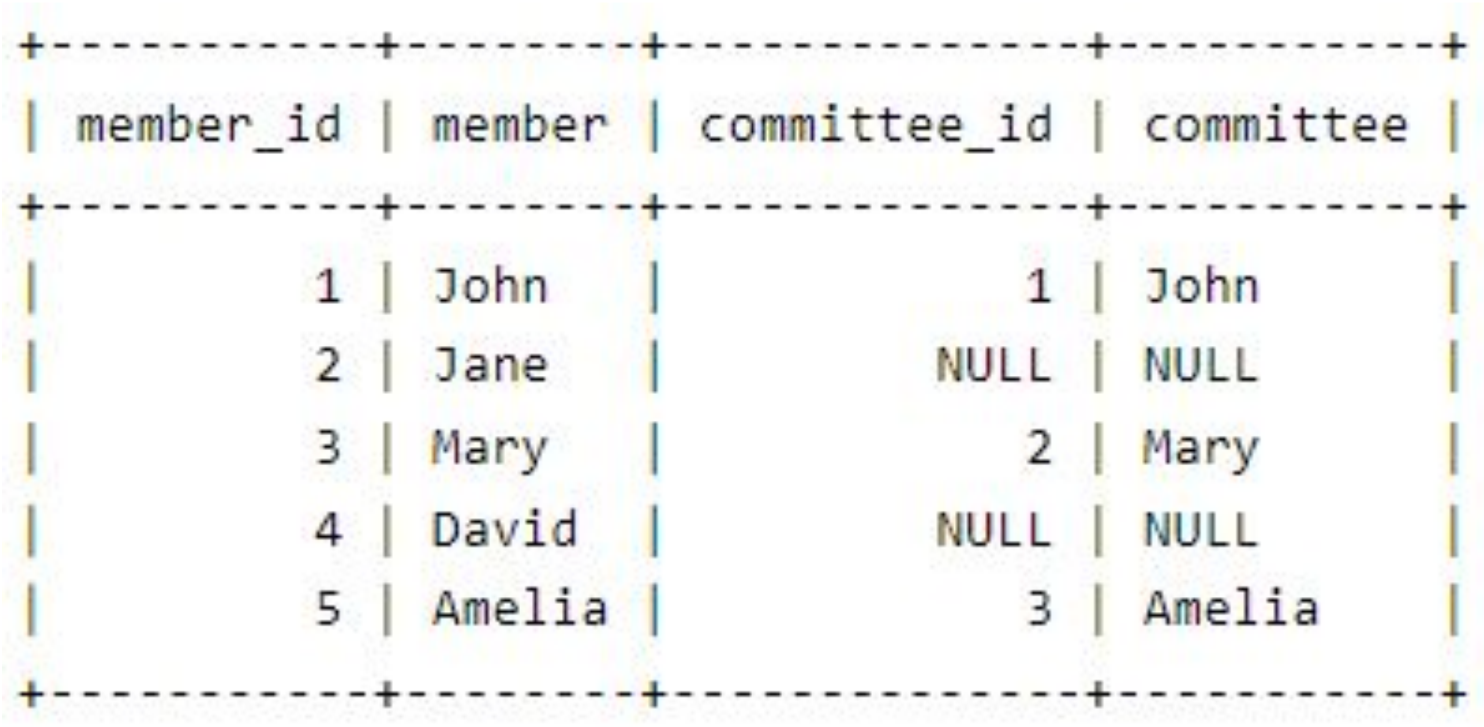


```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         1 | John   |            1 | John      |
|         3 | Mary   |            2 | Mary      |
|         5 | Amelia |            3 | Amelia    |
+-----------+--------+--------------+-----------+
```

LEFT JOIN clause

- The left join selects all data from the left table whether there are matching rows exist in the right table or not.
- In case there are no matching rows from the right table found, the left join uses NULLs for columns of the row from the right table in the result set.

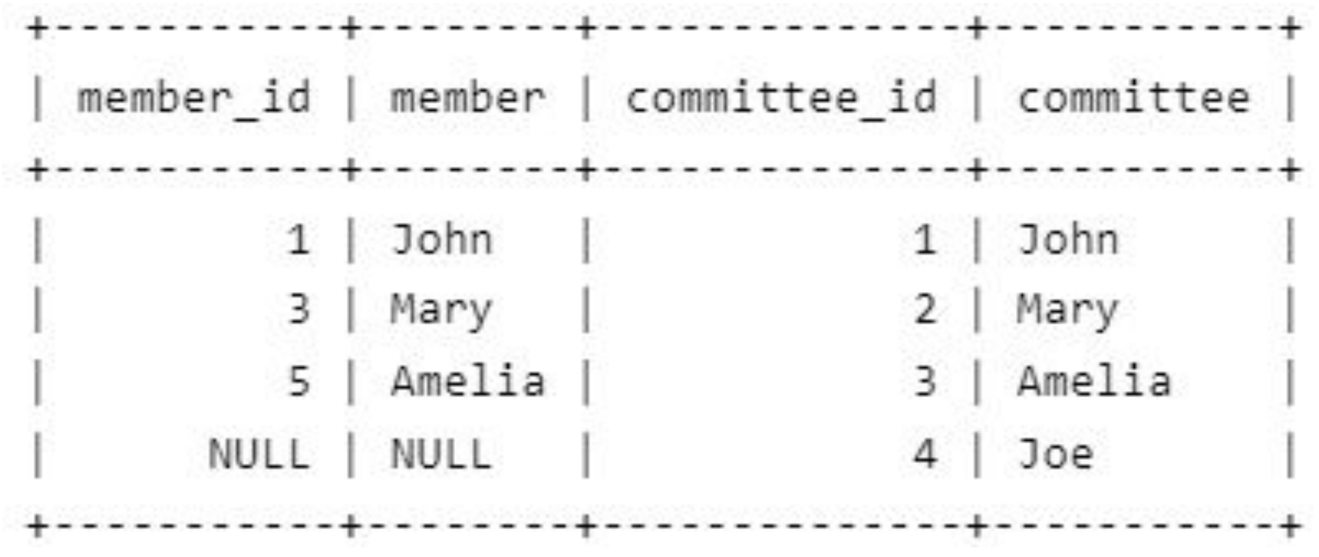
```
SELECT
    m.member_id, m.name AS member, c.committee_id,
    c.name AS committee
FROM members m LEFT JOIN committees c USING(name);
```



```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         1 | John   |            1 | John      |
|         2 | Jane   |         NULL | NULL      |
|         3 | Mary   |            2 | Mary      |
|         4 | David  |         NULL | NULL      |
|         5 | Amelia |            3 | Amelia    |
+-----------+--------+--------------+-----------+
```

Right Join

```
SELECT
   m.member_id, m.name AS member, c.committee_id,
   c.name AS committee
FROM members m
RIGHT JOIN committees c on c.name = m.name;
```



```
+-----------+--------+--------------+-----------+
| member_id | member | committee_id | committee |
+-----------+--------+--------------+-----------+
|         1 | John   |            1 | John      |
|         3 | Mary   |            2 | Mary      |
|         5 | Amelia |            3 | Amelia    |
|      NULL | NULL   |            4 | Joe       |
+-----------+--------+--------------+-----------+
```

The Rename Operation

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

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

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

- select name, course id
  from instructor, teaches
  where instructor.ID= teaches.ID;

- We can rewrite the query as
  - select name as instructor name, course id
    from instructor, teaches
    where instructor.ID= teaches.ID;

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, make set comparisons, and determine set cardinality by nesting subqueries in the where clause

Nested Subqueries

- The nesting can be done in the following SQL query

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  as follows:
  - A_i can be replaced be 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.

- A common use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality.

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

- 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);
```

Set Comparison

- "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';
- Rewrite the query in a form that resembles closely our formulation of the query in English.

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

Set Comparison

- The construct **> all** corresponds to the phrase “greater than all.”

- 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’);
```

“Find the departments that have the highest average salary.”

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= all (select avg (salary)
                            from instructor
                            group by dept_name);
```

Test for Empty Relations

- SQL includes a feature for testing whether a subquery has any tuples in its result.
- The **exists** construct returns the value true if the argument subquery is nonempty.
- 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

Use of not-exists clause

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

Test for the Absence of Duplicate Tuples

- SQL includes a Boolean function for testing whether a subquery has duplicate tuples in its result.
- The **unique** construct returns the value true if the argument subquery contains no duplicate tuples
- 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.
- The key concept applied here is that any select-from-where expression returns a relation as a result and, therefore, can be inserted into another select-from-where anywhere that a relation can appear.
- 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;
```

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;
```

Complex queries using with clause

- Find all departments where the total salary is greater than the average of the total salary at all departments
- We can create an equivalent query without the with clause, but it would be more complicated and harder to understand.

```
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;
```

Subqueries in **Select** clause

Scalar subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

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

- Runtime error if subquery returns more than one result tuple

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

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’);

Deletion contd..

- 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)

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*);

Insertion contd….

- 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*);

Updates

- If a salary increase is to be paid only to instructors with a salary of less than $70,000

```
update instructor
set salary = salary * 1.05
where salary < 70000;
```

- “Give a 5 percent salary raise to instructors whose salary is less than average”

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
 from instructor);
```

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;

  - In such cases the order of query is important
  - Can be done better using the **case** statement (next slide)

Updates

- SQL provides a case construct that we can use to perform both updates with a single update statement, avoiding the problem with the order of 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
```

- Consider an update where we set the tot cred attribute of each student tuple to the sum of the credits of courses successfully completed by the student.
- We assume that a course is successfully completed if the student has a grade that is neither 'F' nor null.

Updates with scalar subqueries

- Recompute and update tot_creds value for all students

```
  update student S
    set tot_cred = (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
```

Exercise

Find all instructors earning the highest salary (there may be more than one with the same salary).

Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.

Delete all courses that have never been offered (i.e., do not occur in the *section* relation).

Views

- It may be required that all users are not allowed to see the entire set of relations in the database
- security considerations may require that only certain data in a relation be hidden from a user.
- we may wish to create a personalized collection of “virtual” relations that is better matched to a certain user’s intuition of the structure of the enterprise.
- It is possible to compute and store the results of queries and then make the stored relations available to users
- But updates done in original table will not be reflected in these stored queries

Views

- Instead, SQL allows a “virtual relation” to be defined by a query, and the relation conceptually contains the result of the query
- The virtual relation is not precomputed and stored but instead is computed by executing the query whenever the virtual relation is used.

View

- View can be created in SQL by using the **create view** command
- To define a view, we must give the view a name and must state the query that computes the view.

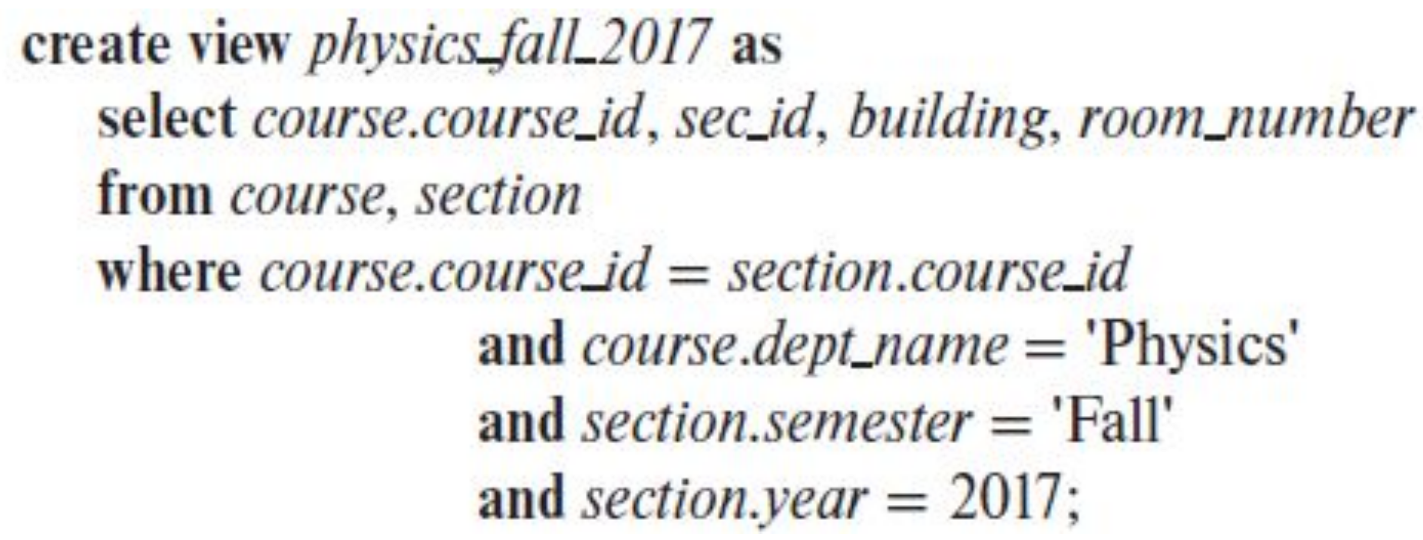
Syntax: *create view v as <query expression>;*

- E.g. Consider the clerk who needs to access all data in the instructor relation, except salary.
- The clerk should not be authorized to access the instructor relation
- View relation **faculty** can be made available to the clerk, with the view defined as follows:

create view faculty as select ID, name, dept name from instructor;

- The database system stores the query expression associated with the view relation.
- View names may appear in a query any place where a relation name may appear

- To create a view that lists all course sections offered by the Physics department in the Fall 2017 semester with the building and room number of each section:



```
create view physics_fall_2017 as
    select course.course_id, sec_id, building, room_number
    from course, section
    where course.course_id = section.course_id
                    and course.dept_name = 'Physics'
                    and section.semester = 'Fall'
                    and section.year = 2017;
```

*create **view** departments_total_salary(dept_name, total_salary) as select dept_name, sum (salary) from instructor group by dept name;*

- Here attribute names for the view are explicitly given
- views are usually implemented as follows:
  - When we define a view, the database system stores the definition of the view itself, rather than the result of evaluation of the query expression that defines the view.
  - Wherever a view relation appears in a query, it is replaced by the stored query expression.
  - Thus, whenever we evaluate the query, the view relation is recomputed.

Using View

- Using the view physics fall 2017, we can find all Physics courses offered in the Fall 2017 semester in the Watson building by writing:

  select course id from physics fall 2017

  where building = 'Watson';

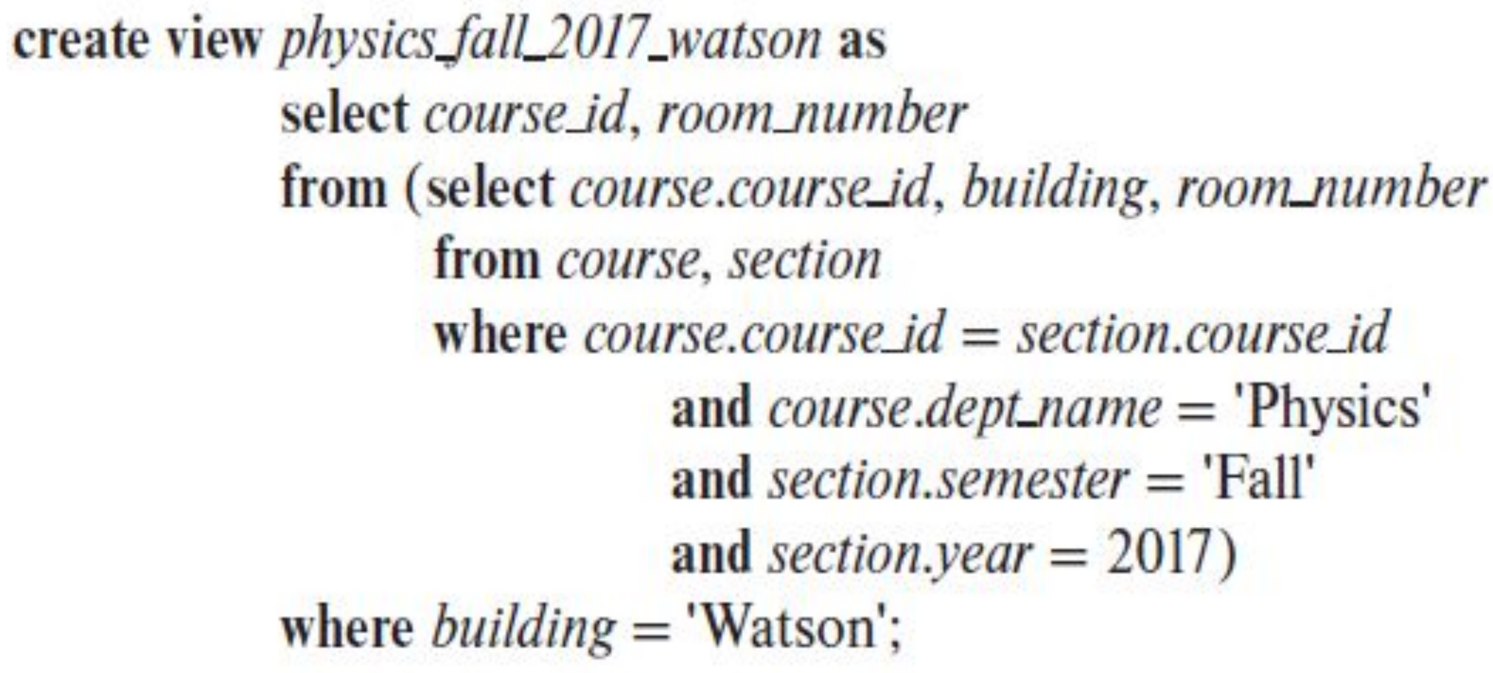
- One view may be used in the expression defining another view.
- For example, we can define a view physics_fall_2017_watson that lists the course ID and room number of all Physics courses offered in the Fall 2017 semester in the Watson building as:

  create view physics_fall_2017_watson as

  select course id, room number from physics_fall_2017

  where building = 'Watson’;

  where physics_fall_2017 is itself a view relation. This is equivalent to:



```
create view physics_fall_2017_watson as
        select course_id, room_number
        from (select course.course_id, building, room_number
              from course, section
              where course.course_id = section.course_id
                    and course.dept_name = 'Physics'
                    and section.semester = 'Fall'
                    and section.year = 2017)
        where building = 'Watson';
```

Views

- In general, an SQL view is said to be updatable (i.e., inserts, updates, or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:
  - The from clause has only one database relation
  - The select clause contains only attribute names of the relation and does not have any expressions, aggregates, or distinct specification.
  - Any attribute not listed in the select clause can be set to null; that is, it does not have a not null constraint and is not part of a primary key.
  - The query does not have a group by or having clause.

- Consider the view definition:

```
create view history instructors as
select *
from instructor
where dept name = 'History’;
```

- User can insert the tuple

  ('25566', 'Brown', 'Biology', 100000)

- This tuple can be inserted into the instructor relation, but it would not appear in the history instructors view since it does not satisfy the selection imposed by the view.

- By default, SQL would allow the above update to proceed.
- However, views can be defined with a with check option clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view’s where clause condition, the insertion is rejected by the database system.
- Updates are similarly rejected if the new value does not satisfy the where clause conditions.
- An alternative, and often preferable, approach to modifying the database through a view is to use the **trigger** mechanism