Practice Exercise 2.6

a. \( \sigma_{year \geq 2009}(takes) \bowtie student \)

b. \( \sigma_{year \geq 2009}(takes \bowtie student) \)

c. \( \Pi_{ID, name, course_id}(student \bowtie takes) \)
Write relational algebra expressions to:

a. Find the names of all employees who live in city “Miami”.
b. Find the names of all employees whose salary is greater than $100,000.
c. Find the names of all employees who live in “Miami” and whose salary is greater than $100,000.
Chapter 3: Introduction to SQL
History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on our particular system.
Data Definition Language (DDL)

Allows specification of information about relations:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints (e.g. NOT NULL)
- Other information such as
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.
Domain Types 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 $n$ digits to the right of decimal point.
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n)**. Floating point number, with user-specified precision of at least $n$ digits.
- More are covered in Ch.4.
Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length \( n \).
- **varchar(n)**. Variable length character strings, with user-specified maximum length \( n \).

How does SQL compare strings of different lengths?
Standard SQL says: Spaces are added to the shorter one until strings have same length.

- Not always implemented correctly!

How about **Unicode**?

- **nvarchar**

- But many implementations allow UTF-8 in varchar
Domain Types 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 $n$ digits to the right of decimal point.

P does not include the sign, nor the decimal point!
Domain Types 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 \( n \) digits to the right of decimal point.
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n)**. Floating point number, with user-specified precision of at least \( n \) digits.

Each of the above also supports the **NULL** value

- Sometimes NULL is prohibited!
Remember the university DB

- **Student**
  - ID
  - name
  - dept_name
  - tot_cred

- **Course**
  - course_id
  - title
  - dept_name
  - credits

- **Department**
  - dept_name
  - building
  - budget

- **Advisor**
  - s_id
  - i_id

- **Section**
  - course_id
  - sec_id
  - semester
  - year

- **Classroom**
  - building
  - room_no
  - capacity

- **Time Slot**
  - time_slot_id
  - day
  - start_time
  - end_time

- **Teaches**
  - ID
  - course_id
  - sec_id
  - semester
  - year

- **Prereq**
  - course_id
  - prereq_id

- **Instructor**
  - ID
  - name
  - dept_name
  - salary
CREATE TABLE

■ Syntax:

\[
\text{CREATE TABLE } r \left( A_1 \ D_1, A_2 \ D_2, \ldots, A_n \ D_n, \\
\text{(integrity-constraint}_1), \\
\ldots, \\
\text{(integrity-constraint}_k) \right)
\]

- \( r \) 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:

\[
\text{CREATE TABLE } \text{instructor} \left( \\
ID \ \text{char}(5), \\
name \ \text{varchar}(20) \ \text{not null}, \\
department_name \ \text{varchar}(20), \\
salary \ \text{numeric}(8,2) \right)
\]

\text{insert into } \text{instructor} \ \text{values} \left( \text{‘10211’}, \ \text{‘Smith’}, \ \text{‘Biology’}, \ 66000 \right);
\text{insert into } \text{instructor} \ \text{values} \left( \text{‘10211’}, \ \text{null}, \ \text{‘Biology’}, \ 66000 \right);
Integrity Constraints in Create Table

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

Example: Declare `dept_name` as the primary key for `department`.

```sql
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 declaration on an attribute automatically ensures not null
Based on this Example, write the CREATE TABLE for **instructor**

```sql
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
)
```
QUIZ: Integrity Constraints

Write CREATE TABLEs for student and takes

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)
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.
QUIZ: Integrity Constraints

Write the CREATE TABLE for course

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
)
CREATE TABLE course (  
course_id varchar(8) primary key,
title varchar(50),
department_name varchar(20),  
credits numeric(2,0),  
foreign key (department_name) references department) ;

Note: Primary key declaration can be combined with attribute declaration as shown.
drop table student
- Deletes the table and its contents

drop from student
- Deletes all contents of table, but retains 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 many databases
3.3 Basic Query Structure

- The SQL data-manipulation language (DML) provides the ability to query information, and insert, delete and update tuples.

- 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 of an SQL query is a relation.
The SELECT clause

- It lists 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:
  
  ```sql
  select name
  from instructor
  ```

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

- 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:

    ```sql
    SELECT distinct dept_name
    FROM instructor
    ```

- The keyword all specifies that duplicates not be removed (seldom used, since it’s the default):

    ```sql
    SELECT all dept_name
    FROM instructor
    ```
The SELECT clause

- 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

- Specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.

- Find all instructors in Comp. Sci. dept with salary > 80000

  ```sql
  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`

  ```sql
  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:** \textit{instructor X teaches}

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
<th>teaches.ID</th>
<th>course_id</th>
<th>sec_id</th>
<th>semester</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2009</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2009</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>12121</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>33456</td>
<td>Goh</td>
<td>Physics</td>
<td>85000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2009</td>
</tr>
</tbody>
</table>

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

```sql
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

```sql
select section.course_id, semester, year, title
from section, course
where section.course_id = course.course_id and dept_name = 'Comp. Sci.'
```
Find the names of all students who are taking a class this semester in the SCIENCE building
QUIZ

What are the three integrity constraints we have covered? Give examples of how each is used.
QUIZ

What are the three integrity constraints we have covered? Give examples of how each is used.

- NOT NULL
- PRIMARY KEY
- FOREIGN KEY
QUIZ

True or false:

By default, in SQL a table may contain duplicate tuples.
QUIZ

True or false:

By default, in SQL a table may contain **duplicate** tuples.
QUIZ

How can we control duplicates in SQL queries?
QUIZ

How can we control duplicates in SQL queries?

```sql
mydb=# SELECT DISTINCT * FROM one;
     col1
-----
     1
     3
     2
(3 rows)
```
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;`

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
<th>course_id</th>
<th>sec_id</th>
<th>semester</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2009</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2009</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2009</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
<td>HIS-351</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
<td>CS-101</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
<td>CS-319</td>
<td>1</td>
<td>Spring</td>
<td>2010</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
<td>BIO-101</td>
<td>1</td>
<td>Summer</td>
<td>2009</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
<td>BIO-301</td>
<td>1</td>
<td>Summer</td>
<td>2010</td>
</tr>
</tbody>
</table>
Natural Join Example

List the names of instructors along with the course ID of the courses that they taught.

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

- `select name, course_id` 
  `from instructor natural join teaches;`
Beware of unrelated attributes with same name which get equated incorrectly

Example: List the names of instructors along with the titles of courses that they teach

- Incorrect version (makes course.dept_name = instructor.dept_name)
  
  ```sql
  select name, title
  from instructor natural join teaches natural join course;
  ```

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

- Another correct version
  
  ```sql
  select name, title
  from (instructor natural join teaches)
  join course using(course_id);
  ```
QUIZ: Natural Join

Use the natural join to write queries for the following:

- Find the names of customers who have (at least) an account
- Find the names of customers who have an account at the “Uptown” branch
- Find the names of customers who have a loan at a branch with assets greater than $1,000,000
The RENAME Operation

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.’`

- The keyword **AS** is optional and may be omitted
  - `instructor as T ≡ instructor T`
Use the relational-algebra operations defined to find all the accounts with a balance smaller than the balance of at least one other account.

**Hint: join the table with itself!**
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 include the substring “dar”:

  ```sql
  select name
  from instructor
  where name like '%dar%'
  ```

- Match the string “100 %”:

  ```sql
  like '100 \%' escape '\'
  ```
### String Operations

- 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.
Find all course names that:

- Begin with “Computer”
- Have the substring “Computer” anywhere in them
- Have the substring “Computer” or “computer” anywhere in them
- Have the substring “Comp” or “comp” anywhere in them
- Have the substring “comp%_” anywhere in them
QUIZ: Find the loan number of all loans made in cities that start with ‘S’ and end in ‘ville’:

Hint: Use join and string operations.
Ordering the Display of Tuples

- List in alphabetic order the names of all instructors:
  ```sql
  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 (inclusive)

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

- Tuple comparison

  ```sql
  select name, course_id
  from instructor AS I, teaches AS T
  where (I.ID, dept_name) = (T.ID, 'Biology')
  ```
WHERE Clause Predicates

- SQL includes an **IN** operator

  - Example: Find the names and salaries of all instructors named ‘Jones’, ‘James’ or ‘Julian’

  ```sql
  SELECT name
  FROM instructor
  WHERE salary IN ('Jones', 'James', 'Julian')
  ```
Quiz: WHERE Clause Predicates

List all the WHERE predicates we learned:
Quiz: WHERE Clause Predicates

List all the WHERE predicates we learned:

- BETWEEN
- Tuple comparison
- IN
Quiz: WHERE Clause Predicates

Write a query to return the **names** of all students who have exactly 50, 60, 70, or 80 credit hours total

- Without a WHERE predicate
- With a WHERE predicate
Quiz: WHERE Clause Predicates

Write a query to return the names of all students who have anywhere between 50 and 80 credit hours total

- Without a WHERE predicate
- With a WHERE predicate
Quiz: WHERE Clause Predicates

Write a query that uses tuple comparison to return the names of all students who have taken a class in 2014.

Use the previous example:

```sql
select name, course_id
from instructor AS I, teaches AS T
where (I.ID, dept_name) = (T.ID, 'Biology')
```
Quiz: WHERE Clause Predicates

Write a query that uses tuple comparison to return the names of all students who have taken a class in 2014.

```sql
SELECT S.name
FROM student AS S, takes AS T
WHERE (T.ID, T.year) = (S.ID, 2014)
```
Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010

\[
\text{(select course\_id from section where sem = 'Fall' and year = 2009)}
\]
union
\[
\text{(select course\_id from section where sem = 'Spring' and year = 2010)}
\]

- Find courses that ran in Fall 2009 and in Spring 2010

\[
\text{(select course\_id from section where sem = 'Fall' and year = 2009)}
\]
intersect
\[
\text{(select course\_id from section where sem = 'Spring' and year = 2010)}
\]

- Find courses that ran in Fall 2009 but not in Spring 2010

\[
\text{(select course\_id from section where sem = 'Fall' and year = 2009)}
\]
except
\[
\text{(select course\_id from section where sem = 'Spring' and year = 2010)}
\]
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** \( s \)
QUIZ: Set operations

Find the names of customers who have an account at the “Uptown” branch or a loan from the “Downtown” branch
Find the account with the maximum balance … without using aggregation functions, i.e. max().

Now we can finally write the complete solution!
Hint: Use EXCEPT
Null Values

- It is possible for tuples to have a null value, denoted by \textit{null}, for some of their attributes.

- \textit{null} signifies an unknown value or that a value does not exist.

- The result of any arithmetic expression involving \textit{null} is \textit{null}.
  - Example: $5 + \text{null}$ returns null.

- The predicate \textbf{is null} can be used to check for null values.
  - Example: Find all instructors whose salary is null.

  \begin{verbatim}
  select name
  from instructor
  where salary is null
  \end{verbatim}
Null Values and Three Valued Logic

- Any comparison with null returns unknown
  - Example: $5 < \text{null}$ or $\text{null} <> \text{null}$ or $\text{null} = \text{null}$

- Three-valued logic using the truth value unknown:
  - OR: $(\text{unknown or true}) = \text{true}$,
    $(\text{unknown or false}) = \text{unknown}$
    $(\text{unknown or unknown}) = \text{unknown}$
  - AND: $(\text{true and unknown}) = \text{unknown}$,
    $(\text{false and unknown}) = \text{false}$,
    $(\text{unknown and unknown}) = \text{unknown}$
  - NOT: $(\text{not unknown}) = \text{unknown}$
  - “$P$ is unknown” evaluates to true if predicate $P$ evaluates to unknown
Null Values and Three Valued Logic

Three-valued logic using the truth value unknown:

- OR: 
  \[ (\text{unknown} \text{ or } \text{true}) = \text{true}, \]
  \[ (\text{unknown} \text{ or } \text{false}) = \text{unknown} \]
  \[ (\text{unknown} \text{ or } \text{unknown}) = \text{unknown} \]

- AND: 
  \[ (\text{true and unknown}) = \text{unknown}, \]
  \[ (\text{false and unknown}) = \text{false}, \]
  \[ (\text{unknown and unknown}) = \text{unknown} \]

- NOT: 
  \[ (\text{not unknown}) = \text{unknown} \]

Write the truth table for three-valued OR!
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Null Values and Three Valued Logic

The result of where clause predicate is treated as false if it evaluates to unknown.

This means that, in SQL, we prefer to err on the side of caution, retaining only those tuples that surely make the predicate true.
Write the truth table for three-valued AND!
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A AND B</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
3.7 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 Examples

- Find the average salary of instructors in the Computer Science department
  
  ```sql
  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
  
  ```sql
  select count (distinct ID)
  from teaches
  where semester = 'Spring' and year = 2010;
  ```

- Find the number of tuples in the `course` relation
  
  ```sql
  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;
```

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

<table>
<thead>
<tr>
<th>dept_name</th>
<th>avg_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>77333</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>Finance</td>
<td>85000</td>
</tr>
<tr>
<td>History</td>
<td>61000</td>
</tr>
<tr>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>Physics</td>
<td>91000</td>
</tr>
</tbody>
</table>
Aggregate Functions – GROUP BY

Attributes in select clause outside of aggregate functions must appear in group by list

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

/* incorrect */

```sql
/* incorrect */
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```
QUIZ: Aggregate Functions

Write queries to find the following:

- The highest budget of all departments
- The lowest number of credit hours of any student
- The highest budget of a department in each building
- The lowest number of credit hours of any student in each department
QUIZ: Aggregate Functions

Write a query to find the lowest number of credit hours of any student in the “CS” department.
QUIZ: Aggregate Functions

Write a query to find the highest budget of any department.

student
- ID
- name
- dept_name
- tot_cred

department
- dept_name
- building
- budget

advisor
- s_id
- i_id
QUIZ: Aggregate Functions

Write a query to find the name of the department with the highest budget.
QUIZ: Aggregate Functions

Write a query to find the **name** of the department with the highest budget.

- Does this work? Why not?

```sql
SELECT dept_name, MAX(budget)
FROM department
```
QUIZ: Aggregate Functions

Write a query to find the **name** of the department with the highest budget.

- Does this work? Why not?

```sql
SELECT dept_name, MAX(budget)
FROM department
```

Attributes in `select` clause outside of aggregate functions must appear in `group by` list.
Aggregate Functions – Having Clause

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

```sql
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.
QUIZ: Aggregate Functions

Write queries to find the following:

- The **names** of the department whose students have an average total credit of at least 6 credit hours
Null Values and Aggregates

- Total all salaries

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

  - Above statement ignores null amounts
  - Result is null if all amounts are null

- All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes

- What if the attribute has only null values?
  - count returns 0
  - all other aggregates return null
3.8 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.
Nested Example: IN and NOT IN

- Find courses offered in Fall 2009 and in Spring 2010

```sql
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

```sql
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);
```
QUIZ: Now we can solve this!

Write a query to find the name of the department with the highest budget.

(Hint: Start with the inner query!)
Extra-credit QUIZ

Write a **nested query** to find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

Hint: Use the schema diagram of the University_DB (handout).
Write a nested query to 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);
```
Write a nested query to find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101.

```sql
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 code above is simply to illustrate SQL features.
3.8 Review: Nested Subqueries

Are used for:

- Determining membership (or lack thereof) in a set:
  - IN
  - NOT IN

Example:

```sql
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);
```

Outer query

Inner query
Nested Subqueries

Are used for:

- Determining membership (or lack thereof) in a set:
  - IN
  - NOT IN

- Comparing tuples against sets:
  - <comparison> SOME
  - <comparison> ALL
Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

\[
\text{select distinct } T.\text{name} \\
\text{from instructor as } T, \text{ instructor as } S \\
\text{where } T.\text{salary} > S.\text{salary} \text{ and } S.\text{dept\_name} = \text{'Biology'};
\]

Same query using \textbf{> some} clause

\[
\text{select name} \\
\text{from instructor} \\
\text{where salary} > \text{SOME}(\text{select salary} \\
\text{from instructor} \\
\text{where dept\_name} = \text{'Biology'});
\]
Formal definition of SOME clause

\[ F \text{ <comp> some } r \iff \exists t \in r \text{ such that } (F \text{ <comp> } t) \]

<comp> can be: \(<, \leq, >, =, \neq\)

\[
\begin{array}{c|c|c|c|c}
  & 0 & 5 & 6 \\
\hline
5 \text{ < some} & \text{true} & \text{true} & \text{true} \\
\end{array}
\]

(read: 5 < some tuple in the relation)

\[
\begin{array}{c|c|c|c|c}
  & 0 & 5 & 6 \\
\hline
5 \text{ < some} & \text{false} & \text{false} & \text{false} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
  & 0 & 5 & 6 \\
\hline
5 \text{ = some} & \text{true} & \text{true} & \text{true} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
  & 0 & 5 & 6 \\
\hline
5 \text{ \neq some} & \text{true} & \text{true} & \text{true} \\
\end{array}
\]

\(\text{(5 \neq \text{some}) \equiv \text{in}}\)

However, \(\text{(5 \neq \text{some}) \not\equiv \text{not in}}\)
QUIZ: SOME

Write a nested query to find the departments with budgets lower than some (at least one) departments in the Watson building.
Set Comparison: ALL clause

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');
```
Formal definition of ALL clause

\[ F \ <\ \text{comp}> \ \textbf{all} \ r \iff \forall t \in r \ (F \ <\ \text{comp}> \ t) \]

\[
\begin{array}{c|c|c}
\hline
0 & 5 & 6 \\
\hline
5 & 6 & 10 \\
\hline
4 & 5 & \\
\hline
4 & 6 & \\
\hline
\end{array}
\]

\( (5 < \text{all} 5) = \text{false} \)

\( (5 < \text{all} 6) = \text{true} \)

\( (5 = \text{all} 5) = \text{false} \)

\( (5 \neq \text{all} 6) = \text{true} \) (since \( 5 \neq 4 \) and \( 5 \neq 6 \))

\( (\neq \text{all}) \equiv \text{not in} \)

However, \( (= \text{all}) \neq \text{in} \)
Write a nested query to find the departments with budgets lower than all departments in the Watson building.
Nested Subqueries

Are used for:

- Determining membership (or lack thereof) in a set:
  - IN
  - NOT IN
- Comparing a tuple against a set:
  - <comparison> SOME
  - <comparison> ALL
- Testing if a relation (table, set) is empty (or not):
  - NOT EXISTS
  - EXISTS

exists returns true iff the argument subquery is nonempty:

- exists  \( r \iff r \neq \emptyset \)
- not exists  \( r \iff r = \emptyset \)
Yet another way to write the query

“Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”:

```sql
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);
```
Yet another way to write the query

“Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”:

```sql
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**
Write an SQL expression that is true iff relation (table) A is contained in relation (table) B
Write an SQL expression that is true iff relation A is contained in relation B

NOT EXISTS (A EXCEPT B)
Find the names of students who have taken all courses offered in the Biology department.

Hint: There is no course they have not taken!
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));
```

What does this inner query mean?
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));
```

Is the inner query correlated with the outer?
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));
```

What would the query return if the correlation condition were missing?
Find all students who have taken all courses offered in the Biology department.

\[
\text{select distinct } S.ID, S.name \\
\text{from student as } S \\
\text{where NOT EXISTS ( (select course_id } \\
\text{from course } \\
\text{where dept_name = ’Biology’) } \\
\text{except } \\
\text{ (select } T.course_id } \\
\text{from takes as } T \\
\text{where } S.ID = T.ID));
\]

- \textit{Note: Cannot write this query using } = \textit{all} or any of its variants, b/c it’s not a \textit{set}-based comparison!
Nested Subqueries

Are used for:

- Determining membership (or lack thereof) in a set:
  - \texttt{IN}
  - \texttt{NOT IN}

- Comparing tuples against sets:
  - \texttt{<comparison> SOME}
  - \texttt{<comparison> ALL}

- Testing if a relation is empty (or not):
  - \texttt{NOT EXISTS}
  - \texttt{EXISTS}

- Testing for no duplicate tuples
  - \texttt{UNIQUE}

- Note: \texttt{UNIQUE} of an empty set is \texttt{true}!
Test for Absence of Duplicate Tuples

Find all courses that were offered at most once in 2009:

```sql
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);
```
QUIZ: UNIQUE

Find the names of students who have taken COSC4401 at least once.
Find the names of students who have taken COSC4401 at least once.

```
SELECT S.ID, S.name
FROM student AS S
WHERE UNIQUE (  
  SELECT T.course_id
  FROM takes AS T
  WHERE S.ID = T.ID  AND
  T.course_id = 'COSC4401'
)
```
QUIZ: UNIQUE

Find the names of students who have taken COSC4401 only once.

```
SELECT  S.ID, S.name
FROM     student AS S
WHERE UNIQUE (  
   SELECT  T.course_id
   FROM     takes AS T  
   WHERE     S.ID = T.ID AND
             T.course_id = 'COSC4401'  
)
```
All subqueries we’ve seen so far are in the WHERE clause, i.e. they are used to filter the tuples. SQL allows subqueries in all clauses of the SELECT statement:

- SELECT
- FROM
- WHERE
- GROUP BY
- HAVING
- ORDER BY
3.8.5 Subqueries in the FROM clause

Here the subquery is used as a **data source** for the outer query.

Find the average instructors’ salaries of those departments where the average salary is greater than $42,000.

```sql
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: we do not need to use the **having** clause.
Subqueries in the FROM clause

Find the average instructors’ salaries of those departments where the average salary is greater than $42,000.

Another way to write above query:

```sql
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;
```
Yet another way to write it: **lateral** clause

```sql
select name, salary, avg_salary
from instructor I1,
    lateral (select avg(salary) as avg_salary
           from instructor I2
           where I2.dept_name = I1.dept_name);
```

Lateral clause permits later part of the `from` clause (after the lateral keyword) to access **correlation variables** from the earlier part.

Note: lateral is part of the SQL standard, but is not supported on many DBMSs:

- Some, such as SQL Server, offer alternative syntax.
- PostgreSQL has it!
WITH clause

Provides a way of defining a temporary view 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 budget
from department, max_budget
where department.budget = max_budget.value;
```
Complex Queries using WITH clause

- WITH clause is very useful for writing complex queries
- Supported by most DBMSs, with minor syntax variations
- Find all departments where the total salary is greater than the average of the total salary at all departments

```sql
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;
```
What does this nested query return?

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

If a single value is expected, SQL will extract the scalar value from the relation/table returned by the subquery

- Can be used in the SELECT clause:

```sql
SELECT dept_name,
    (select count(*)
    from instructor
    where department.dept_name = instructor.dept_name
    ) AS num_instructors
FROM department;
```
Scalar Subquery

- Can also be used in WHERE or HAVING clauses

```sql
select name
from instructor
where salary * 12 >
  (select budget from department
   where department.dept_name = instructor.dept_name)
```
### Scalar Subquery

- Can also be used in WHERE or HAVING clauses

```sql
select name
from instructor
where salary * 12 >
  (select budget from department
  where department.dept_name = instructor.dept_name)
```

**QUIZ:** Write a (nested) query that returns the names of instructors whose salary is less than the average salary in their department.
Scalar Subquery

As seen here, scalar subqueries are not limited to aggregate functions:

```sql
select name
from instructor
where salary * 12 >
  (select budget from department
   where department.dept_name = instructor.dept_name)
```

QUIZ: Can you imagine a case where the subquery returns more than one tuple?

Runtime error if subquery returns more than one result tuple.
3.9 Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating 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');`
Delete all instructors whose salary is less than the average salary of all 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)

**Aggregate functions in subqueries are not correlated!**
Insertion

- Add a new tuple to \textit{course}
  \begin{verbatim}
  insert into course
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
  \end{verbatim}

- or equivalently
  \begin{verbatim}
  insert into course (course_id, title, dept_name, credits)
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
  \end{verbatim}

- Add a new tuple to \textit{student} with \textit{totcreds} set to null
  \begin{verbatim}
  insert into student
  values ('3003', 'Green', 'Finance', null);
  \end{verbatim}
Insertion

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

```sql
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

```sql
insert into table1 select * from table1
```

would cause problems, if *table1* did not have any primary key defined.
Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise

- Write two `update` statements:
  ```sql
  update instructor
  set salary = salary * 1.03
  where salary > 100000;

  update instructor
  set salary = salary * 1.05
  where salary <= 100000;
  ```

- The order is important! (Why?)

- Can be done better using the `case` statement (next slide)
update instructor
    set salary = case
        when salary <= 100000 then salary * 1.05
        else salary * 1.03
    end

What does this update accomplish?
Recompute and update tot_creds value for all students

```sql
update student S
set tot_cred = ( select sum(credits)
from takes natural join course
where 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:

```sql
case
  when sum(credits) is not null then 
    sum(credits)
  else 0
end
```
Problem 3.10

Employee (employee_name, street, city)
Works(employee_name, company_name, salary)
Company(company_name, city)
Manages(employee_name, manager_name)

- Modify the DB so that “Jones” now lives in “Newtown”
- Give all managers of “First Bank Corporation” a 10% raise
- Give all managers of “First Bank Corporation” a 10% raise, unless the salary becomes > $100,000; in such cases, give only a 3% raise
Homework for Ch.3, due Thu, Feb.19:

--End of chapter Practice Exercise: 3.8
--End of chapter Exercises: 3.11, 12, 16
--Table with all SQL constructs from ch.3, each used in an example (OK if example is from text)
Schema for Bank database (fig. 3.19)