Chapter 3: Introduction to SQL

Database System Concepts, 7th Ed.

Chapter 3: Introduction to SQL

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

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
  - SQL:1999, SQL:2003, SQL:2008, SQL:2011, SQL:2016,SQL:2023
- 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 your particular system.

Data Definition Language

The SQL **data-definition language (DDL)** allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- Integrity constraints
- And as we will see later, also 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 Chapter 4.

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) **not null,**
  dept_name **varchar**(20),
  salary **numeric**(8,2))

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

Integrity Constraints in Create Table

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

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

.

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

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

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 not be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

Some constraints can be with attribute definition

- **create table** *course* (
  course_id **varchar**(8) **primary key**,
  title **varchar(**50),
  dept_name **varchar**(20),
  credits **numeric**(2,0),
  foreign key *(dept_name*) **references** *department)* ;
  - Primary key declaration can be combined with attribute declaration as shown above

Drop and Alter Table Constructs

- **drop table** *student*
  - Deletes the table and its contents
- **delete from** *student*
  - It's not a DDL, just to see the difference with drop table
  - 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

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 $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  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.

Basic Query Structure – Cont.

- **select** *A1, A2, .. An*
  from *r1, r2, …, rm*
  where P

  is equivalent to the following expression in **multiset** relational algebra

  $$\Pi_{A1,\ ..,\ An}(\sigma_{P}(r1 \times r2 \times .. \times rm))$$

- Different from normal relational algebra, SQL does not eliminate duplicates. Because in real application environment, there are sometimes duplicate information.
- SQL names are case insensitive. E.g. *Name* ≡ *NAME* ≡ *name*

Select distinct

- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the names of all departments with instructor, and remove duplicates

    select distinct *dept_name*
    from *instructor*

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

    select all *dept_name*
    from *instructor*

Select * and calculation of results

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

  select '437'

The where Clause

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

```
select name
from instructor
where dept_name = 'Comp. Sci.'  and salary > 80000
```

- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

The from Clause

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

  select *
  from *instructor, teaches*

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

Cartesian Product: *instructor X teaches*

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

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

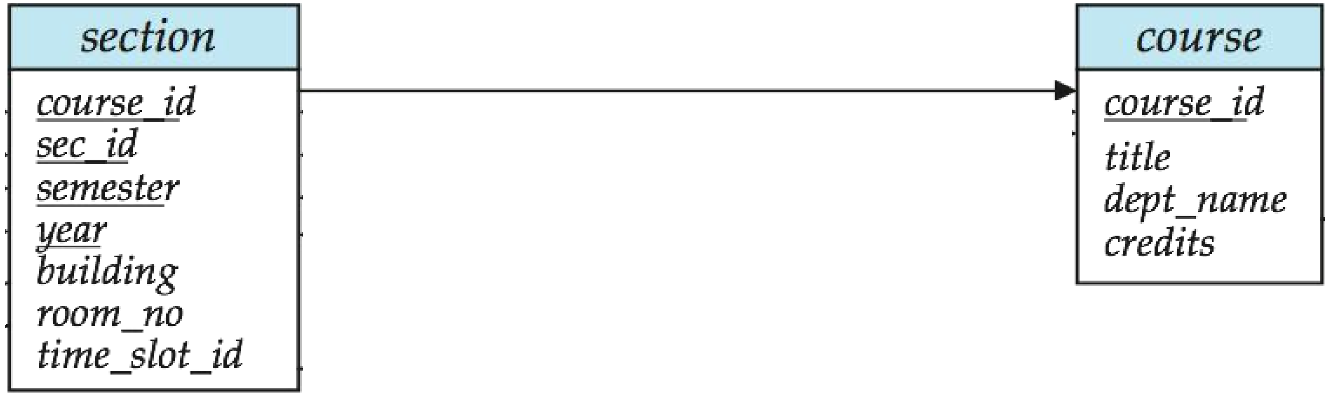
Cartesian Product with conditions- Joins

- Find instructors names and the course ID of the courses they taught.

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

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

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



Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column
- **select** *name, course_id*
  from *instructor* **natural join** *teaches*;
- $\Pi_{name,course\_id}$ (*instructor* $\bowtie$ *teaches*)

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

Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the the titles of courses that they teach
  - Incorrect version (makes course.dept_name = instructor.dept_name)
    - **select** *name*, *title*
      from *instructor* **natural join** *teaches* **natural join** *course*;
  - Correct version
    - **select** *name*, *title*
      from *instructor* **natural join** *teaches*, *course*
      where *teaches*.*course_id* = *course*.*course_id*;

The Rename Operation

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

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

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

- Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.
  - **select distinct** *T. name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary > S.salary* **and** *S.dept_name = ‘Comp. Sci.’*

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

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

  - Keyword **as** must be omitted in Oracle

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** '\'

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

- 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*

Specific Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

1. $\boldsymbol{\sigma_\theta(r_1)}$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
2. $\boldsymbol{\Pi_A(r)}$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
3. $\boldsymbol{r_1 \times r_2}$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

- Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

  $r_1 = \{(1, a)\ (2,a)\}$ $\quad r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be

  $\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$

- SQL duplicate semantics:

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

  is equivalent to the *multiset* version of the expression:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

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

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  union
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

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

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  intersect
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

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

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  except
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

Set Operations

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

  Suppose a tuple occurs m times in r and n times in s, then, it occurs:
  - $m + n$ times in r **union all** s
  - $\min(m,n)$ times in r **intersect all** s
  - $\max(0, m - n)$ times in r **except all** s

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

    select *name*
    from *instructor*
    where *salary* **is null**

Null Values and Three Valued Logic

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

Aggregate Functions

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

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

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count** (**distinct** *ID*)
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2010
- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;
  - Note: departments with no instructor will not appear in result

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

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

Aggregate Functions – Group By – Cont.

- More generally, the non-aggregated attributes in the **select** clause may be a subset of the **group by** attributes, in which case the equivalence is as follows:

  select *A1*, **sum**(*A3*)
  from *r1, r2, …, rm*
  where P
  group by *A1, A2*

  is equivalent to the following expression in multiset relational algebra

  $$\Pi_{A1,sumA3}(_{A1,A2}\mathcal{G}_{\mathbf{sum}(A3)\ \mathbf{as}\ sumA3}(\sigma_P(r1 \times r2 \times .. \times rm)))$$

- /* erroneous queries*/
  - **select** *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;
  - **select** *ID*, *max(salary)* **from** *instructor*

Aggregate Functions – Having Clause

- For each department, find the average salaries of those instructors with salary greater than 30000, and then, output the department name and the average value which is greater than 42000

select *dept_name*, **avg** (*salary*)
from *instructor*
where *salary* > 30000
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

Equivalent to :

$$\Pi_{dept\_name,\ avg\_salary}(\sigma_{avg\_salary\ >\ 42000}({}_{dept\_name}\ \mathcal{G}_{avg(salary)\ as\ avg\_salary}\ (\sigma_{salary\ >\ 30000}\ (\text{instructor})))$$

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

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

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- If subquery is in where clause, it's a kind of predicate term involving sets. E.g. 1) whether the value of an attribute is in a subset; 2) whether the value of an attribute is greater than some of or all the values in a subset; 3) whether the subset is empty or not; 4) whether the subset cardinality <= 1 or not.
- If subquery is in from clause, it's just a temporary relation.
- We can also use subquery to divide a complex query into several different steps, as the assignment operation in relational algebra.

Nested Subqueries – 'In'

- Find instructor names who are advisors of students

```
select name from instructor where ID in (select i_id  from advisor)

select name from instructor, advisor where  ID = i_id
```

- Find instructor names who are not advisors of students
- 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 number of courses offered in Fall 2009 and in Spring 2010

Set Comparison – 'Some'

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

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

- Same query using > **some** clause

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

Definition of Some Clause

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

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

(read: 5 < some tuple in the relation)

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

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

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

(= **some**) ≡ **in**
However, (≠ **some**) ≡ **not in**

Set Comparison – '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');
```

Definition of all Clause

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

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

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

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

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

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

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

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

($\neq$ **all**) $\equiv$ **not in**

However, (= **all**) $\equiv$ $\cancel{\textbf{in}}$

Test for Empty Relations

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

Nested Subqueries – 'Exists'

- Find instructor names who are advisors of students

```
select name from instructor where exists (select * from
advisor where i_id = instructor.ID)
```

- Find instructor names who are not advisors of students

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct 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 T.course_id = S.course_id );
```

- *instructor.ID* and *S.course_id* are attributes that are not in the relation of the subquery, they are called **correlation variables**

Use ‘Not Exists’ to Realize Division

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

```
select 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 T.ID = S.ID));
```

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
  - (Evaluates to "true" on an empty set)
- 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 R.course_id= T.course_id
                    and R.year = 2009);
```

Subqueries in the From Clause

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

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

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

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

Subqueries in the From Clause (Cont.)

- And yet another way to write it: **lateral** clause

```
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 database systems; some databases such as SQL Server offer alternative syntax

With Clause

- The **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 dept_name
from department, max_budget
where department.budget = max_budget.value;
```

- Similar to assign operation of relational algebra

Complex Queries using With Clause

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

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

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- E.g. **select** *name*
  from *instructor*
  where *salary * 10 >*
  (**select** *budget* **from** *department*
  where *department.dept_name* = *instructor.dept_name*)
- Runtime error if subquery returns more than one result tuple
- Scalar subquery does not conform to concepts of relational algebra. In relational algebra, all the result of a query is a relation. For the above question, you can also write a query based on traditional relational algebra. But it is more complicated.
- For convenience, scalar subqueries are frequently used in practice.
- E.g. Find all departments with the maximum budget (not using ‘with clause’)

select *dept_name*
from *department*
where *budget* = (**select max**(*budget*) **from** *department*)

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

Modification of the Database – Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

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

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

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

Deletion (Cont.)

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

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

Insert One Row

- 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*);

Insert – get data from other relation

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

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

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

Modification of the Database – Updates

- Increase salaries of instructors by 5%
  - **update** *instructor*
        set *salary* = *salary* * 1.05
- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise
  - Write two **update** statements:

    update *instructor*
    set *salary* = *salary* * 1.03
    where *salary* > 100000;
update *instructor*
    set *salary* = *salary* * 1.05
    where *salary* <= 100000;
  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

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

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
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

End of Chapter 3
Exercise 8,9,10,11,15

Database System Concepts, 7^{th} Ed.

Advanced SQL Features**

- Create a table with the same schema as an existing table:

  create table *temp_account* **like** *account*

Figure 3.02

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

Figure 3.03

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

Figure 3.04

name
Katz
Brandt

Figure 3.05

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

Figure 3.07

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

Figure 3.08

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

Figure 3.09

course_id
CS-101
CS-347
PHY-101

Figure 3.10

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

Figure 3.11

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

Figure 3.12

course_id
CS-101

Figure 3.13

course_id
CS-347
PHY-101

Figure 3.16

dept_name	*count*
Comp. Sci.	3
Finance	1
History	1
Music	1

Figure 3.17

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