

Chapter 3: Introduction to SQL

**Edited by Radhika Sukapuram. Original slides by
Database System Concepts, 6th Ed.
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use**



Outline

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



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: 2016
- 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 domain 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.



Domain Types in SQL

- **char(*n*)**. Fixed length character string, with user-specified length *n*. Spaces are appended where required.
- **varchar(*n*)**. Variable length character strings, with user-specified maximum length *n*. No spaces are appended.
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(*p,d*)**. Fixed point number, with user-specified precision of *p* digits, with *d* of *p* digits to the right of decimal point. (ex., **numeric(3,1)**, allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **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 later lectures.



char and varchar

- When two chars are compared, extra blank spaces are appended to the shorter string
- When a char and a varchar are compared, adding extra length to varchar is system dependent
- Use varchar always



Create Table Construct

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

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

- 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, \dots, A_n)
- **foreign key** (A_m, \dots, 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 declaration on an attribute automatically ensures **not null**
SQL prevents any update that violates an integrity constraint



And a Few More Relation Definitions

- **create table student (**

<i>ID</i>	varchar(5),
<i>name</i>	varchar(20) not null,
<i>dept_name</i>	varchar(20),
<i>tot_cred</i>	numeric(3,0),
primary key (<i>ID</i>),	
foreign key (<i>dept_name</i>) references department;	

- **create table takes (**

<i>ID</i>	varchar(5),
<i>course_id</i>	varchar(8),
<i>sec_id</i>	varchar(8),
<i>semester</i>	varchar(6),
<i>year</i>	numeric(4,0),
<i>grade</i>	varchar(2),
primary key (<i>ID, course_id, sec_id, semester, year</i>),	
foreign key (<i>ID</i>) references student,	
foreign key (<i>course_id, sec_id, semester, year</i>) 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



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 existing 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

- A typical SQL query has the form:

```
select A1, A2, ..., An  
from r1, r2, ..., rm  
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

- The **select** clause 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:

```
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 (Cont.)

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

```
select distinct dept_name  
from instructor
```

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

```
select all dept_name  
from 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



The select Clause (Cont.)

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

```
select *
from instructor
```

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

```
select '437'
```

- Result 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 (Cont.)

- 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.



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.
- For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).



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

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



Meaning of an SQL query

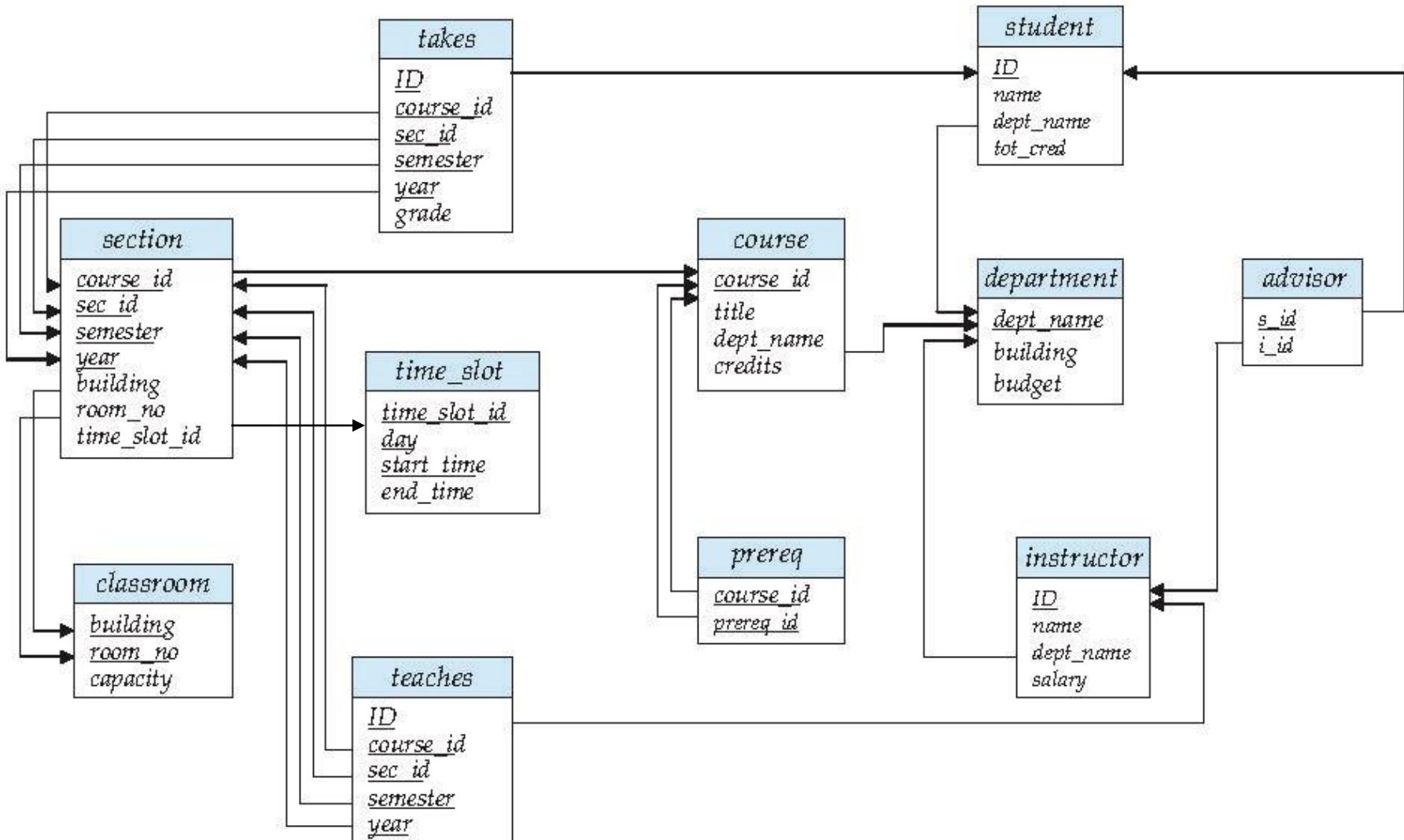
```
select A1, A2, ..., An  
from r1, r2, ..., rm  
where P
```

- Generate a Cartesian product of the relations listed in the **from** 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, \dots, t_m into a single tuple t
 - Add t into the result relation
 - Apply the predicates specified in the **where** clause on the result of Step 1.
 - For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the **select** clause.

Note: A real implementation will not execute the query as above



Schema Diagram for University Database





Examples

- Find the names of all instructors who have taught some course and the course_id. Use *instructor* and *teaches*
 - **select** *name, course_id*
from *instructor , teaches*
where *instructor.ID = teaches.ID*
- Find the names of all instructors in the Art department who have taught some course and the course_id
 - **select** *name, course_id*
from *instructor , teaches*
where *instructor.ID = teaches.ID and instructor.dept_name = 'Art'*



Natural Join

- Natural join:

```
select A1, A2, ..., An  
from r1 natural join r2 natural join ... natural join rm  
where P
```

- “For all instructors in the university who have taught some course, find their names and the course ids of all the courses they have taught”
- instructor(ID, name, dept_name, salary), teaches(ID, course_id, section_id, semester, year)

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

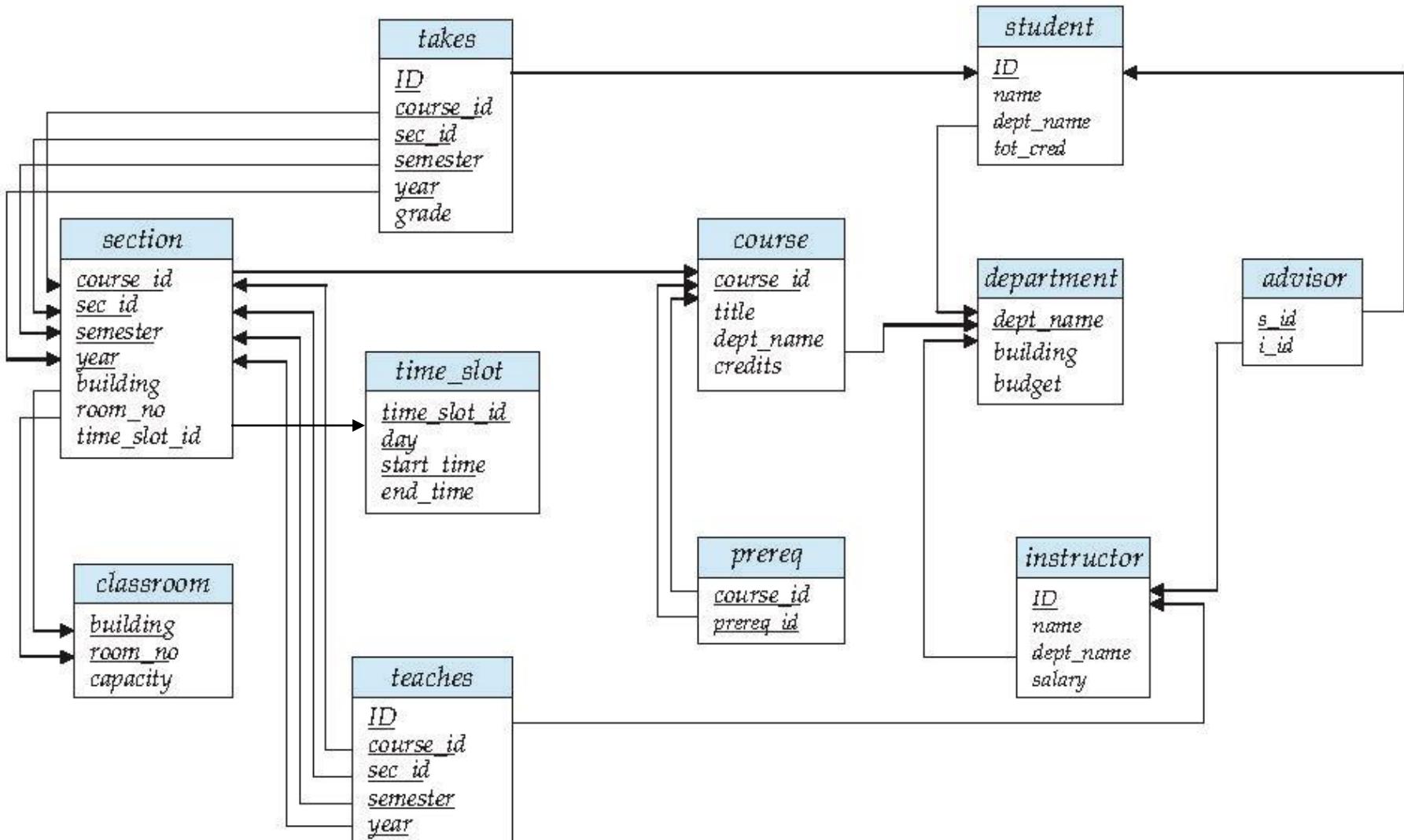
OR

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

A Natural join is associative and commutative



Schema Diagram for University Database





Natural Join cont.

- List the names of instructors along with the courses that they teach

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

instructor(ID, name, dept_name, salary)

teaches(ID, course_id, section_id, semester, year)

course(course_id, title, dept_name, credits)

Does the query below compute the same result ?

```
select name, title  
from instructor natural join teaches natural join course
```

No!

To avoid equating attributes wrongly,

```
select name, title  
from (instructor natural join teaches) join course using (course_id);
```



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
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*

Note: T is called a correlation name, table alias, correlation variable or tuple variable.



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

- List in alphabetic order the names of all instructors

```
select distinct name  
from instructor  
order by name
```

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



Where Clause Predicates

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



section

<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>building</i>	<i>room_number</i>	<i>time_slot_id</i>
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A



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 (Cont.)

- Find the salaries of all instructors that are less than the largest salary.
 - **select distinct** *T.salary*
from *instructor as T, instructor as S*
where *T.salary < S.salary*
- Find all the salaries of all instructors
 - **select distinct** *salary*
from *instructor*
- Find the largest salary of all instructors.
 - **(select “second query”)**
except
(select “first query”)



Set Operations (Cont.)

- 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: $5 + \text{null}$ 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

- Three logical values – *true*, *false*, *unknown*
- 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 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*



Where predicate with null

- Example: Find all instructors whose salary is greater than 5000.

```
select name  
from instructor  
where salary > 500
```

Suppose a record has salary = null.

What value will the predicate return for that record ?

Will it be treated as true or false ?

As false



null: select distinct, set operations

- Eliminating duplicate tuples

```
select distinct name  
from instructor  
where salary > 500
```

Suppose two records have name as null

Will both the records be retained ?

No, only one will be retained – treatment different from predicates

- Same treatment for set operations



Aggregate Functions

- These functions operate on the (collection) 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;**



instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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



Aggregate Functions – Group By

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

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000



Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
 - /* erroneous query */

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```
 - Which id must be output corresponding to a group (dept_name) is not known



Aggregate Functions – Having Clause

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

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

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



Order of evaluation of aggregate queries

select A_1, A_2, \dots, A_n	5
from r_1, r_2, \dots, r_m	1
where P	2
group by A_1, A_2, \dots, A_n	3
having P_2	4

1. **from** clause is evaluated first to get a relation
2. If a **where** clause is present, its predicate is applied
3. Tuples are then grouped as per **group by** if it exists
4. **having** clause is applied to each group; groups that do not satisfy the clause are removed
5. **select** clause uses tuples of the remaining groups to generate the result



Null Values and Aggregates

- Total all salaries

```
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 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.
- Nesting can be done in the following SQL query

```
select A1, A2, ..., An
  from r1, r2, ..., rm
 where P
```

as follows:

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

$B <\text{operation}> (\text{subquery})$

Where B is an attribute and $<\text{operation}>$ to be defined later.



Subqueries in the Where Clause



Subqueries in the Where Clause

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



section

<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>building</i>	<i>room_number</i>	<i>time_slot_id</i>
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A



Set Membership

- 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);
```

*B <operation> (subquery)
course_id in ...*

- 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);
```

- Evaluate inner query, then outer query



Set Membership (Cont.)

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

teaches(ID, course_id, section_id, semester, year) Note: ID is instructor ID

takes(ID, course_id, section_id, semester, year, grade) Note: ID is student ID

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

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



Set Comparison – “some” Clause

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

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

- Same query using > **some** clause

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



Definition of “some” Clause

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

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$ (read: 5 < some tuple in the relation)

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

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

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

$(= \text{some}) \equiv \text{in}$

However, $(\neq \text{some}) \not\equiv \text{not in}$



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');
```



Definition of “all” Clause

- $F \text{ <comp> all } r \Leftrightarrow \forall t \in r (F \text{ <comp> } t)$

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

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

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

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

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

However, $(= \text{all}) \neq \text{in}$



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** $r \Leftrightarrow r = \emptyset$



Use of “exists” Clause

- 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);
```

- Evaluate **from** clause of outer query, then predicate of outer **where** clause (partially), then select *course_id* of that tuple
 - Evaluate inner query using the *course_id* (that is, *S.course_id*)
 - Do the same for all tuples of the outer query
-
- **Correlation name** – variable *S* in the outer query
 - **Correlated subquery** – the inner query
 - Scoping rules



Use of “not exists” Clause

- Find all students who have taken all courses offered in the Biology department.
- *course(course_id, title, dept_name, credits), takes(ID, course_id, section_id, semester, year, grade)* Note: *ID* is student ID

```
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));
```

12, 13, 14
12, 13, 14, 15

- First nested query lists all courses offered in Biology
 - Second nested query lists all courses a particular student took
-
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$



section

<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>building</i>	<i>room_number</i>	<i>time_slot_id</i>
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A



Test for Absence of Duplicate Tuples

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

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

Note: If no course is offered in 2009, the subquery returns an empty set and unique returns true



Subqueries in the From Clause



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;
```



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.

- Available for using only in later parts of same query

■ 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

```
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 the Select Clause



Scalar Subquery

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

SQL extracts attribute from relation

```
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
- Scalar subquery possible in **select**, **where** and **having** clauses



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');

- **delete** operates only on one relation



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);
```

- 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

student(id, name, dept_name, tot_creds)

insert into student

values ('3003', 'Green', 'Finance', null);



Insertion (Cont.)

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

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

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

Otherwise queries like

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

would cause problems



Updates

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

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

- The order is important
- Can be done better using the **case** statement (next slide)
- Each statement is applied once to each of the tuples in *instructor*



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

student(id, name, dept_name, tot_creds)

takes(ID, course_id, section_id, semester, year, grade)

course(course_id, title, dept_name, credits)

- Recompute and update *tot_creds* value for all students

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

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

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

- What is the purpose?



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. $\sigma_\theta(r_1)$: 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. $\Pi_A(r)$: 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. $r_1 \times r_2$: 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 $t_1 \cdot t_2$ 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 $\Pi_B(r_1) \times r_2$ would be $\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$
- SQL duplicate semantics:

```
select A1, A2, ..., An
```

```
from r1, r2, ..., rm
```

```
where P
```

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

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

End of Chapter 3

**Edited by Radhika Sukapuram. Original slides by
Database System Concepts, 6th Ed.**

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use