Unit 3: Introduction to SQL

Unit 3: Introduction to SQL

Introduction to SQL:

- 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

Intermediate and Advanced SQL:

- Joined Relations
- Views
- Authorization

SQL Query Language

- **Structured Query Language (SQL) SQL** is a very simple, yet powerful, database access language.
- **SQL** is a **non-procedural** language; users describe in **SQL** what they want to do, and the **SQL** language compiler automatically generates a procedure to navigate the database and perform the desired task.

Overview of The SQL Query Language

- IBM developed the Sequel language as part of System R project at the IBM San Jose Research Laboratory in the early 1970s.
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86 (1986)
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
  - SQL:2006
  - SQL:2008 and so on.
- 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.

SQL Server Release History

Version	Year	Release Name
1.0 (OS/2)	1989	SQL Server 1.0 (16 bit)
1.1 (OS/2)	1991	SQL Server 1.1 (16 bit)
4.21 (WinNT)	1993	SQL Server 4.21
6.0	1995	SQL Server 6.0
6.5	1996	SQL Server 6.5
7.0	1998	SQL Server 7.0
-	1999	SQL Server 7.0 OLAP Tools
8.0	2000	SQL Server 2000
8.0	2003	SQL Server 2000 64-bit Edition
9.0	2005	SQL Server 2005
10.0	2008	SQL Server 2008
10.25	2010	SQL Azure DB
10.5	2010	SQL Server 2008 R2
11.0	2012	SQL Server 2012
12.0	2014	SQL Server 2014
13.0	2016	SQL Server 2016
14.0	2017	SQL Server 2017
15.0	2019	SQL Server 2019
Announced	2022	SQL Server 2022

Parts of SQL language

- ***Data-definition language (DDL).*** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- ***Interactive data-manipulation language (DML).*** The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It also includes commands to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.

- ***View definition.*** The SQL DDL includes commands for defining views.
- ***Transaction control.*** SQL includes commands for specifying the beginning and ending of transactions.
- ***Embedded SQL and dynamic SQL.*** Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.
- ***Authorization.*** The SQL DDL includes commands for specifying access rights to relations and views.

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
- 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 *d* of the *p* 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.

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: department

  create table *department* (
  dept_name ***var*char**(20),
  building **varchar**(15)**,**
  budget **numeric**(8,2),
  primary key (dep_name));

Integrity Constraints

- Integrity constraints guard database against accidental damage, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Integrity Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate
- **Foreign key**

Not Null and Unique Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**
- **primary key:**
  - Declare attributes or set of attributes as primary key

    primary key (*ID*),

    primary key (section_ID, course_ID, semester, year)
- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes

    *A*1, *A*2, … *A*m
    form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

- **check** (P)
  where P is a predicate that must be satisfied by every tuple in a relation.
  Example:
  1. Ensure salary of instructor greater than 0:
     check (salary > 0.0),
  2. Ensure that semester value is one of fall, winter, spring or summer: (Enumerated data type)
     check (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer'))
- **create table** *instructor* (
  ID **char**(5),
  name **varchar**(20) **not null,**
  dept_name **varchar**(20),
  salary **numeric**(8,2),
  check (salary > 0.0),
  primary key (*ID*));

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations. If the relation R contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.
- **foreign key**(A_1, A_2, …, A_k) **references** s **on delete cascade on update cascade**

Cascading Actions in Referential Integrity

- **create table** *course (*
  course_id **char**(5),
  title **varchar**(20),
  dept_name **varchar**(20),
  primary key (*course_id*)
  foreign key (*dept_name*) **references** *department)*
- **create table** *course* (
  …
  dept_name **varchar**(20),
  primary key (*course_id*)
  foreign key (*dept_name*) **references** *department*
  on delete cascade
  on update cascade,
  . . .
)
- alternative actions to cascade: **set null**, **set default**

Integrity Constraints in Create Table

- **create table** *instructor* (
  I_ID **varchar**(5),
  name **varchar**(20) **not null,**
  dept_name **varchar**(20),
  salary **numeric**(8,2),
  check (salary > 0.0)
  primary key (*I_ID*),
  foreign key *(dept_name*) **references** *department*
  on delete cascade on update cascade);
- **create table** *student* (
  S_ID **varchar**(5),
  name **varchar**(20) not null,
  dept_name **varchar**(20),
  tot_cred **numeric**(3,0),
  primary key *(S_ID),*
  foreign key *(dept_name*) **references** *department*
  on delete cascade on update cascade);
- **primary key** declaration on an attribute automatically ensures **not null**

And a Few More Relation Definitions

- **create table** *advisor* (
  S_ID **varchar**(5),
  I_ID **varchar(**50),
  primary key *(S_ID),*
  foreign key *(S_ID*) **references** *student on delete cascade on update cascade*
  foreign key *(I_ID*) **references** *instructor on delete cascade on update cascade*);
- **create table** *section* (
  course_id **varchar** (8),
  sec_id **varchar** (8),
  semester **varchar** (6),
  year **numeric** (4,0),
  building **varchar** (15),
  room_number **varchar** (7),
  time slot id **varchar** (4),
  primary key (*course_id*, *sec_id*, *semester*, *year*),
  check (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer'))
  foreign key *(course_id*) **references** *course on delete cascade on update cascade*);

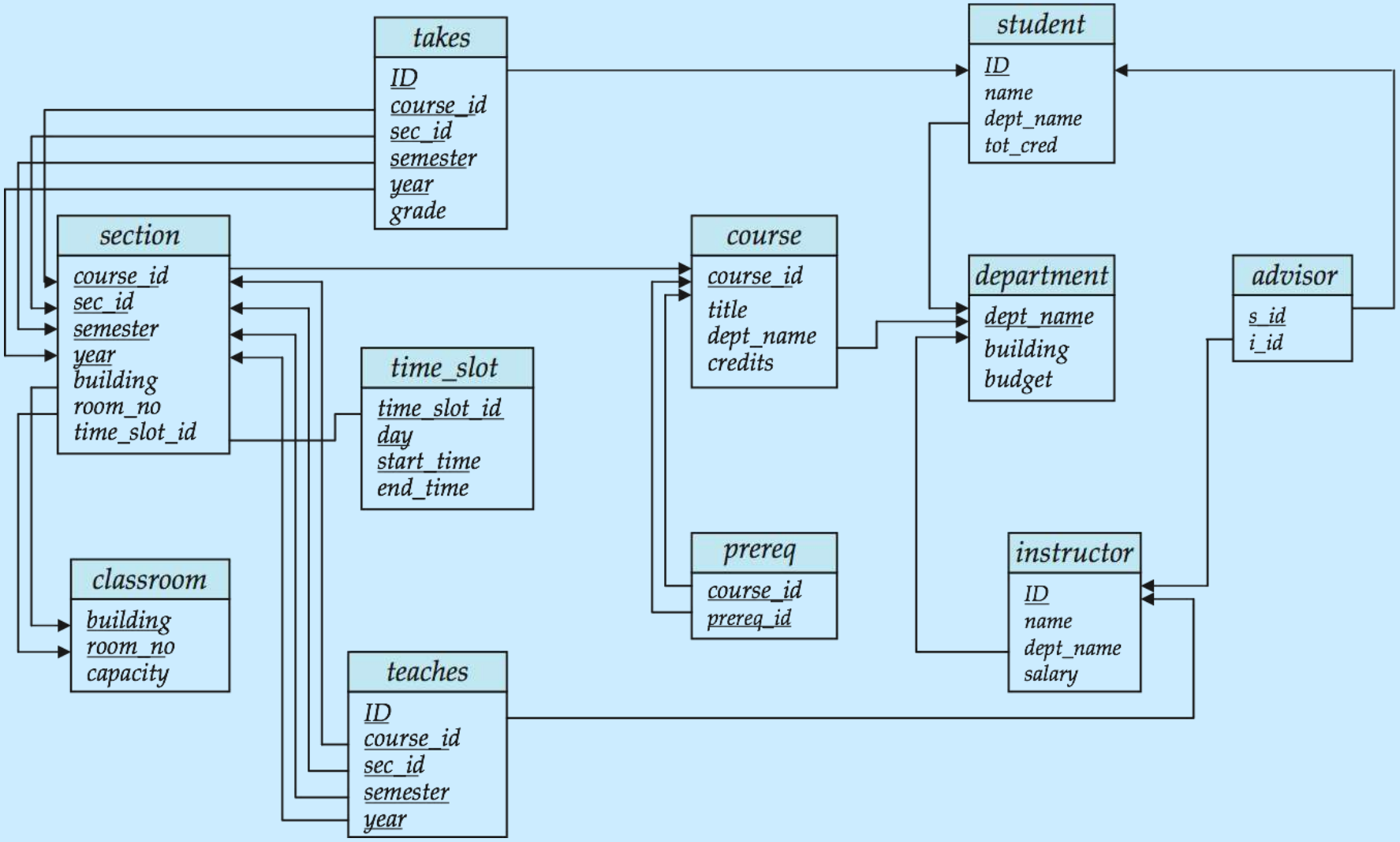
And more still

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

And more still

- **create table** *teaches* (
  I_ID **varchar**(5),
  course_id **varchar**(8),
  sec_id **varchar**(8),
  semester **varchar**(6),
  year **numeric**(4,0),
  primary key *(I_ID, course_id, sec_id, semester, year)* ,
  foreign key *(I_ID*) **references** *instructor on delete cascade on update cascade*
  foreign key (*course_id, sec_id, semester, year*) **references** *section on delete cascade on update cascade*);

Schema Diagram for University Database



Relations in University Database

- department (*dept_name, building, budget).*
- course (*course_id, title, dept_name, credits).*
- instructor (i_*ID, name, dept_name, salary).*
- section (*course id, sec_id, semester, year, building, room_number, time_slot_id).*
- student (*s_ID, name, dept_name, tot_cred).*
- Time_slot (*time_slot_id, day, start_time, end_time)*
- *teaches (i_ID, course id, sec id, semester, year)*
- *takes (s_ID, course id, sec id, semester, year, grade)*
- *prereq (course id, prereq id)*
- *advisor (s ID, i ID)*

Updates to tables

- **Drop Table**
  - **drop table** *r*
  - *Ex:* ***drop table*** *student*
- **Delete**
  - **delete from** *student*
- **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 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 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. (optional)
- The result of an SQL query is a relation.
- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the from clause and its optional.

The select Clause (Queries on Single Relation)

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

select *name*
from *instructor*

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

The select Clause (Queries on Single Relation)

Find the department names of all instructors

select *dept_name* from *instructor*

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

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*

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

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

select all *dept_name*
from *instructor*

The select Clause (Cont.)

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

select *
from *instructor*

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.)

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

  select *ID, name, dept_name, salary * 1.1*
  from *instructor*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is multiplied by 1.1.

The where Clause

- The **where** clause allows us to select only those tuples that must satisfy a specified predicate.
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept with salary > 70000

select *name*
from *instructor*
where *dept_name* = 'Comp. Sci.' **and** *salary* > 70000

name
Katz
Brandt

- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons **(<, ≤, >, ≥, <>, =)** can be applied to results of arithmetic expressions.

Where Clause Predicates

- SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
- 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
- Example: Find the instructor names and the courses they taught for all instructors in the Biology department who have taught some course.
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor*.*ID*, *dept_name*) = (*teaches*.*ID*, 'Biology');

The from Clause (Queries on Multiple Relations)

- 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

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
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…

The from Clause (Queries on Multiple Relations)

- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

  select ∗
  from *instructor, teaches*
  where instructor.ID = teaches.ID

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

Example

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

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

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

Example

Find instructor names and course identifiers for instructors in the Computer Science department

select *name, course_id*
from *instructor, teaches*
where *instructor.ID = teaches.ID* and *dept_name* = 'Comp. Sci.'

name	*Course_id*
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Katz	CS-101
Katz	CS-319
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319

Example

- Retrieve the names of all instructors, along with their department names and department building name.

select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name = department. dept_name

name	*dept_name*	*building*
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Example

- 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.'
```

SQL Query

- In general, the meaning of an SQL query can be understood as follows:
  1. Generate a Cartesian product of the relations listed in the from clause
  2. Apply the predicates specified in the where clause on the result of Step 1.
  3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the select clause.

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

Natural Join

- The ***natural join*** is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.
- The natural-join operation forms a Cartesian product of its two arguments relation, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.
- Natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations and remove duplicate attributes.

Natural Join

- **select** *
  from *instructor* **natural join** *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
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

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*;
- **select** ***A1, A2, . . . , An***
  from r_1 ***natural join*** r_2 ***natural join . . . natural join*** r_m
  where ***P;***

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 salary greater than at least one instructor in 'Biology'.
  - **select distinct** *T. name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary > S.salary* **and** *S.dept_name = 'Biology'*

String Operations

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

String Operations (Cont.)

- 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.
- Patters 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.

String Operations (Cont.)

- SQL expresses patterns by using the **like** comparison operator.
- Find the names of all instructors whose name includes the substring “dar”.

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

- Find the names of all departments whose building name includes the substring ‘Watson’.

```
select dept name
from department
where building like ’%Watson%’;
```

String Operations (Cont.)

- For patterns to include the special pattern characters (that is, % and _), SQL allows the specification of an escape character.
- We define the escape character for a **like** comparison using the **escape** keyword.
- To illustrate, consider the following patterns, which use a backslash (\) as the escape character:
  - **like** 'ab\%cd%' **escape** '\' matches all strings beginning with "ab%cd".
  - **like** 'ab\\cd%' **escape** '\' matches all strings beginning with "ab\cd".

Ordering the Display of Tuples

- SQL offers the user some control over the order in which tuples in a relation are displayed.
- The **order by** clause causes the tuples in the result of a query to appear in sorted order.
- List in alphabetic order the names of all instructors in the physics department

  select *name*
  from *instructor*
  where *dept_name* **=** *'Physics'*
  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**

Ordering the Display of Tuples

- Ordering can be performed on multiple attributes
  - Example: To list the entire instructor relation in descending order of salary. If several instructors have the same salary, we order them in ascending order by name.

```
select *
from instructor
order by salary desc, name asc;
```

Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set-theory operations ∪, ∩, and −.

Set Operations (Cont…)

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

select *course_id*
from *section*
where *sem* = 'Fall' and *year* = 2009

course_id
CS-101
CS-347
PHY-101

select *course_id*
from *section*
where *sem* = 'Spring' and *year* = 2010

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

Set Operations (Cont…)

- 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)

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

- The union operation automatically eliminates the duplicates.

Set Operations (Cont…)

- If we want to retain duplicates, we must write **union all** in place of union.

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

- The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both results
- Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs: *m* + *n* times in *r* **union all** *s*

Set Operations (Cont…)

- 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)

course_id
CS-101

- The intersection operation automatically eliminates the duplicates.

Set Operations (Cont…)

- If we want to retain duplicates, we must write **intersect all** in place of intersect

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

- The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both *r* and *s.*
- Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs min(*m,n)* times in *r* **intersect all** *s*

Set Operations (Cont…)

- 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)

course_id
CS-347
PHY-101

- The except operation automatically eliminates the duplicates.

Set Operations (Cont…)

- If we want to retain duplicates, we must write **except all** in place of except

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

- The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies in *r* minus the number of duplicate copies in *s*, provided that the difference is positive. Otherwise 0.
- Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs max(0, *m* – *n)* times in *r* **except all** *s*

Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set-theory operations ∪, ∩, and −.

- 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)

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**

Aggregate Functions

- ***Aggregate functions*** are functions that take a collection (a set *or multiset) of values* as input and return a single value as output. SQL offers five built-in aggregate functions:

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

Aggregate Functions (Cont.)

- Find average salary of all instructors.
  - **select avg** (*salary*)
    from *instructor;*
- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= ’Comp. Sci.’;
- We can give a meaningful name to the attribute by using the **as** clause as follows
  - **select avg** (*salary*) **as** *avg_salary*
    from *instructor*
    where *dept_name*= ’Comp. Sci.’;
- Retaining duplicates is important in computing an average.

Aggregate Functions (Cont.)

- There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression.
- 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;
- We use the aggregate function **count** frequently to count the number of tuples in a relation.
- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

Aggregate Functions – Group By

- There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause.
- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    from *instructor*
    group by *dept_name*;

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

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

Aggregate Functions – Group By

- Find the number of instructors in each department who teach a course in the Spring 2010 semester.

  select *dept_name*, **count(distinct** ID)
  from *instructor* ***natural join*** *teaches*
  where *semester* **=** *'Spring'* ***and*** *year = 2010*
  group by *dept_name*;

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

Aggregation (Cont.)

- When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the **select** statement without being aggregated are those that are present in the **group by** clause.
- In other words, any attribute that is not present in the **group by** clause must appear only inside an aggregate function if it appears in the **select** clause, otherwise the query is treated as erroneous
- 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*;

Aggregate Functions – Having Clause

- At times, it is useful to state a condition that applies to groups rather than to tuples.
- 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

Aggregate Functions – Having Clause

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

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

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

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

Null Values and Aggregates

- Null values, when they exist, complicate the processing of aggregate operators.
- Assume that some tuples in the *instructor relation have a null value* for *salary*
- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality by nesting subqueries in **where** clause.

Set Membership

- The **in connective tests** for set membership. The **not in** connective tests for the absence of 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);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
        course_id  not in (select course_id
                           from section
                           where semester = 'Spring' and year= 2010);
```

Set Comparison

- Find 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';
```

- The phrase "greater than at least one" is represented in SQL by > ***some.***

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

SQL also allows < ***some, <= some, >= some, = some, and <> some*** comparisons.

Example Query

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.
- The phrase “greater than all” is represented in SQL by > ***all***

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

As it does for **some, SQL also allows < *all*, <= *all*, >= *all*, = *all*, and <> *all*** comparisons.

Example Query

- Find the departments that have the highest average salary.

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

Modification of the Database (DML)

- 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

- We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

  delete from *r*

  where *P*;

- where *P* represents a predicate and *r* represents a relation.
- The **delete** statement first finds all tuples *t* in *r* for which *P*(*t*) is true, and then deletes them from *r*.
- The **where** clause can be omitted, in which case all tuples in *r* are deleted.

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

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

- Delete all instructors with a salary between $13,000 and $15,000.

  delete from *instructor*
  where *salary* **between** 13000 **and** 15000;

- 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*);

  - Problem: as we delete tuples from instructor, 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

- To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted
- Add a new tuple to *course :* course CS-437 in the Computer Science department with title “Database Systems”, and 4 credit hours.

  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);

- or equivalently

  insert into *course* (*title*, *course id*, *credits*, *dept name*)
  values (’Database Systems’, ’CS-437’, 4, ’Comp. Sci.’);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values (’3003’, ’Green’, ’Finance’, *null*);

Insertion

- Suppose that we want to make each student in the Music department who has earned more than 144 credit hours, an instructor in the Music department, with a salary of $18,000.

  insert into *instructor*

  select *ID*, *name*, *dept_name*, 18000

  from *student*

  where *dept name* = ’Music’ **and** *tot cred* > 144

- **insert into** *student*

  select *

  from *student*;

might insert an infinite number of tuples

Updates

- In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used.
- Suppose that annual salary increases are being made, and salaries of all instructors are to be increased by 5 percent.

  update *instructor*

  set *salary*= *salary* * 1.05;
- If a salary increase is to be paid only to instructors with salary of less than $70,000:

  update *instructor*

  set *salary* = *salary* * 1.05

  where *salary* < 70000;
- Give a 5 percent salary raise to instructors whose salary is less than average

  update *instructor*

  set *salary* = *salary* * 1.05

  where *salary* < (**select avg** (*salary*)

  from *instructor*);

Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others 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
```

Intermediate and Advanced SQL

Joined Relations

- **Join operations** take two relations and return another relation as a result.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join.
- The join operations are typically used as subquery expressions in the **from** clause

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.

Join operations – Example

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- **Inner Join**:

  prereq information is missing for CS-315 and
  course information is missing for CS-347

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

Left Outer Join

- **Left outer join :** takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join.
- *course* **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join

- **Right outer join :** takes all tuples in the right relation that did not match with any tuple in the left relation, pads the tuples with null values for all other attributes from the left relation, and adds them to the result of the natural join.
- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join

- **Full outer join: d*oes both the left and right outer join operations,*** padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join.
- *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Joined Relations in SQL – Examples

- To distinguish normal joins from outer joins, normal joins are called **inner joins in** SQL.

  Select *

  From course ***join*** prereq ***using*** (course_id);

  Is equivalent to:

  Select *

  From course ***inner join*** prereq ***using*** (course_id)

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- *course* **inner join** *prereq* **on** *course.course_id = prereq.course_id*

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors ID, name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not part of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

Example Views

- A view of instructors without their salary

  create view *faculty* **as**
  select *ID*, *name*, *dept_name*
  from *instructor*

- Created view can be refer to find information.
- Find all instructors in the Biology department

  select *name*
  from *faculty*
  where *dept_name* = 'Biology'

- Create a view of every department salary totals

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

Views Defined Using Other Views

- Create a view that lists all course sections offered by the Physics department in the Fall 2009 semester with the building and room number of each section
- **create view** *physics_fall_2009* **as**
  select *course*.*course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course*.*course_id* = *section*.*course_id*
  and *course*.*dept_name* = 'Physics'
  and *section*.*semester* = 'Fall'
  and *section*.*year* = '2009';
- Find all Physics courses offered in the Fall 2009 semester in the Watson building
- **select** *course_id*
  from *physics_fall_2009*
  where *building*= 'Watson';

Materialized Views

- Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views.**
- For example, consider the view departments total salary.
- **create view** *departments_total_salary*(*dept_name*, *total_salary*) **as**
  select *dept_name*, **sum** (*salary*)
  from *instructor*
  group by *dept_name*;
- If the above view is materialized, its results would be stored in the database. However, if an instructor tuple is added to or deleted from the instructor relation, the result of the query defining the view would change, and as a result the materialized view's contents must be updated.
- Similarly, if an instructor's salary is updated, the tuple in departments total salary corresponding to that instructor's department must be updated.

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty* **values** (’30765’, ’Green’, ’Music’);

  This insertion must be represented by the insertion of the tuple

  (’30765’, ’Green’, ’Music’, null)

  into the *instructor* relation

Update of a View

- Another problem with modification of the database through views occurs with a view such as:

  create view *instructor_info* ***as***

  select *ID, name, building*

  from *instructor, department*

  where *instructor.dept name= department.dept name;*

- This view lists the *ID, name, and building-name of each instructor in the university.*
- Consider the following insertion through this view:

  insert into ***i****nstructor_info*

  values (’69987’, ’White’, ’Taylor’);

Update of a View

Relations *instructor and department after insertion of tuples.*

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
69987	White	*null*	*null*

instructor

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

department

Authorization

We may assign a user several forms of authorizations on parts of the database. Authorizations on data include::

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Each of these type of authorization is called a **privilege.**

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name>

  to <user list>
- <privilege list> allows granting of several privileges.
- <user list> is:
  - a user-id
  - **public**, all current and future users of the system.
- A user who creates a new relation is given all privileges on that relation automatically.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 the **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement

  grant update (budget) **on** department **to** Amit, Satoshi;
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>
  on <relation name or view name>
  from <user list>

- Example:

  revoke select on *branch* **from** U_1, U_2, U_3

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <user list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- A set of roles is created in the database.
- Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users.
- In university database, examples of roles could include *instructor*, *teaching assistant*, *student*, *dean*, and *department chair*.
- Any authorization that can be granted to a user can be granted to a role. Roles are granted to users just as authorizations are.
- Roles can be created in SQL as follows:
  - **create role** *instructor*;
  - **grant** instructor **to** Amit
- Roles can then be granted privileges just as the users can:
  - **grant select on** *takes* **to** *instructor*;

Roles

- Roles can be granted to users, as well as to other roles, as these statements show:
  - **create role** *dean*;
  - **grant** *dean* **to** Amit;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

End of Chapter 3