# Database Management Systems

## Module 06: Introduction to SQL/1



**Partha Pratim Das**

*Department of Computer Science and Engineering*
*Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar**
**Himadri B G S Bhuyan**
**Gurunath Reddy M**

**Database System Concepts, 6th Ed.**



**Slides marked with 'PPD' are new or edited**

# Week 01 Recap

- **Module 01: Course Overview**
  - Why Databases?
  - KYC: Know Your Course
- **Module 02: Introduction to DBMS/1**
  - Levels of Abstraction
  - Schema & Instance
  - Data Models
  - DDL & DML
  - SQL
  - Database Design
- **Module 03: Introduction to DBMS/2**
  - Database Design
  - Database Engine, Users, Architecture
  - History of DBMS
- **Module 04: Introduction to Relational Model/1**
  - Attribute Types
  - Relation Schema and Instance
  - Keys
  - Relational Query Languages
- **Module 05: Introduction to Relational Model/2**
  - Relational Operations
  - Aggregate Operations

# Module Objectives

- To understand relational query language
- To understand data definition and basic query structure

# Module Outline

- History of SQL
- Data Definition Language (DDL)
- Basic Query Structure (DML)

- **History of SQL**
- Data Definition Language (DDL)
- Basic Query Structure (DML)

# HISTORY OF SQL

# History of Query Language

- 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 (language name became Y2K compliant!)
  - SQL:2003
- 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

- History of SQL
- **Data Definition Language (DDL)**
- Basic Query Structure (DML)

# DATA DEFINITION LANGUAGE (DDL)

# Data Definition Language

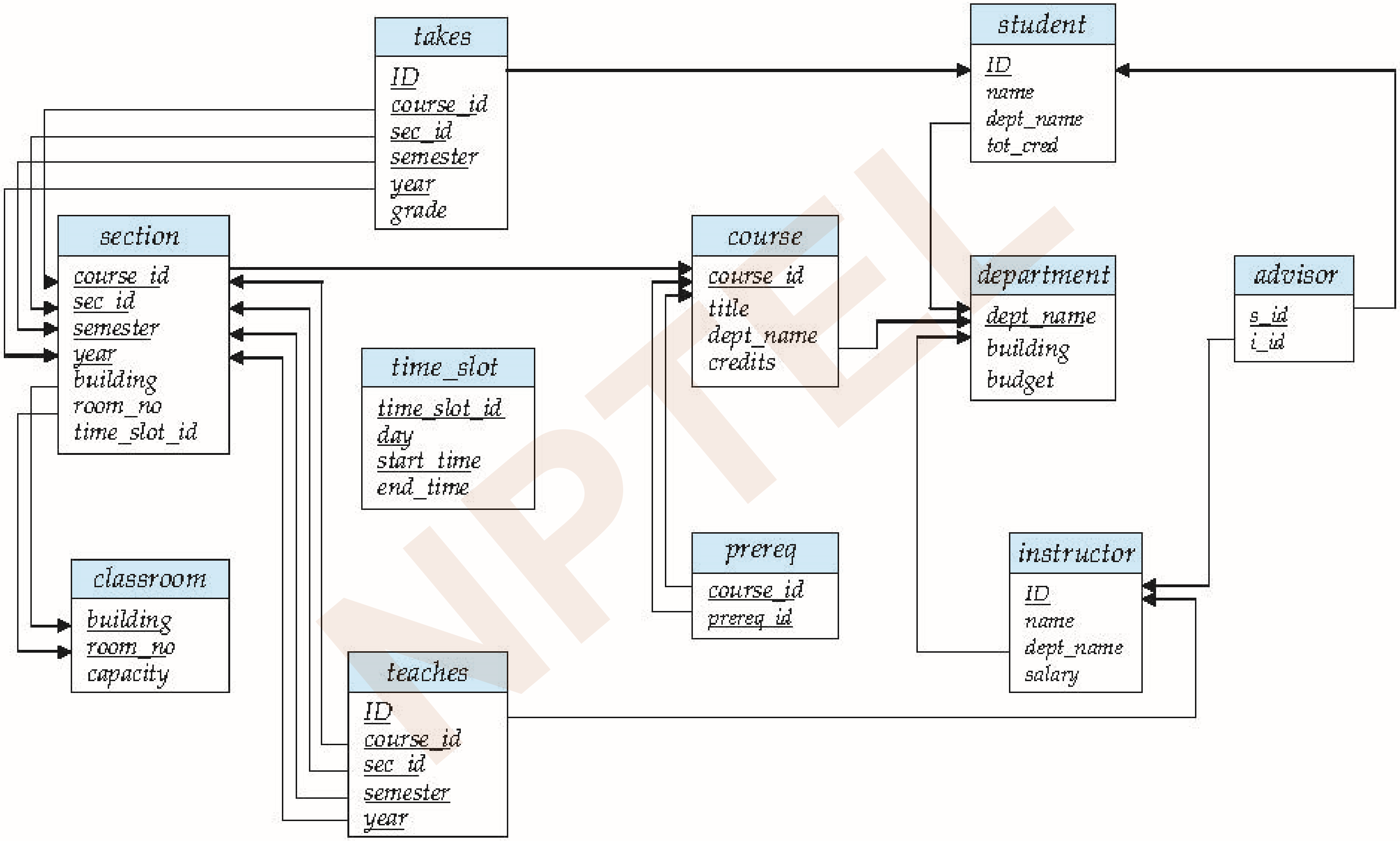
The SQL data-definition language (DDL) allows the specification of information about relations, including:

- 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
  - 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* digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stores 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 Chapter 4

# Schema Diagram for University Database



# Create Table Construct

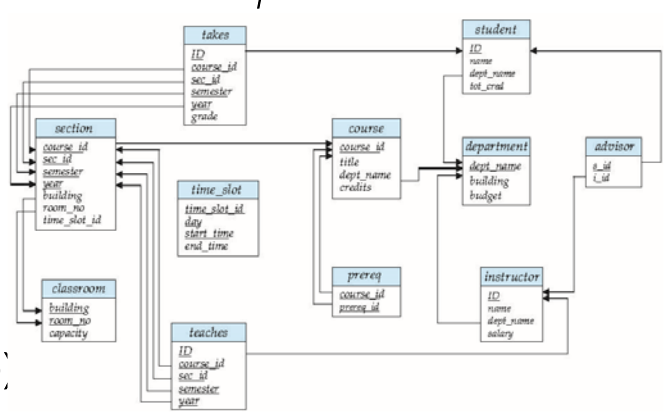
- An SQL relation is defined using the **create table** command:

  **create table** *r* ($A_1$ $D_1$, $A_2$ $D_2$, ..., $A_n$ $D_n$,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *r* is the name of the relation
  - each $A_i$ is an attribute name in the schema of relation *r*
  - $D_i$ is the data type of values in the domain of attribute $A_i$

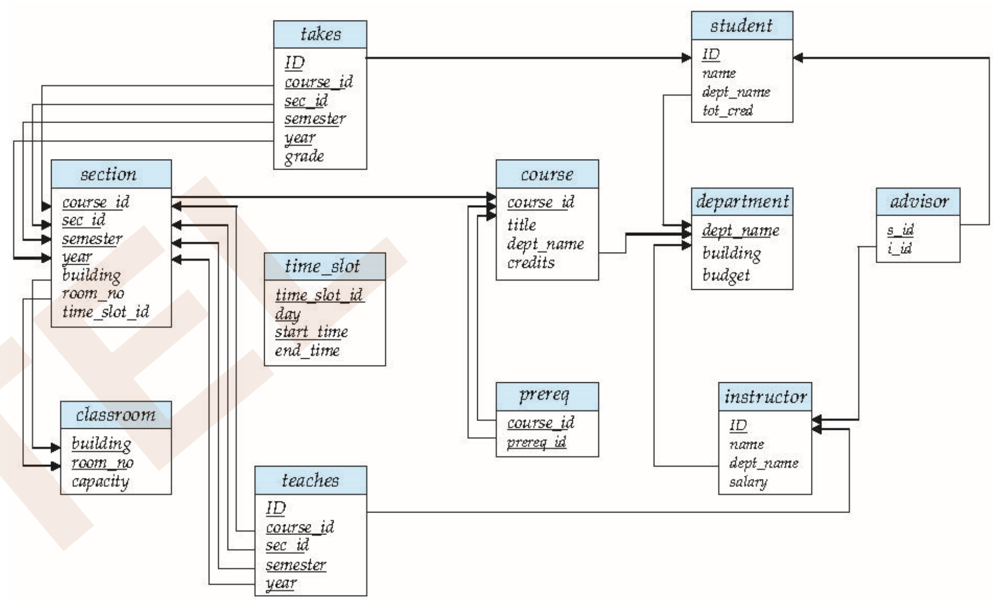
- Example:

  **create table** *instructor* (
  *ID* **char**(5),
  *name* **varchar**(20),
  *dept_name* **varchar**(20),
  *salary* **numeric**(8,2))



# Integrity Constraints in Create Table

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

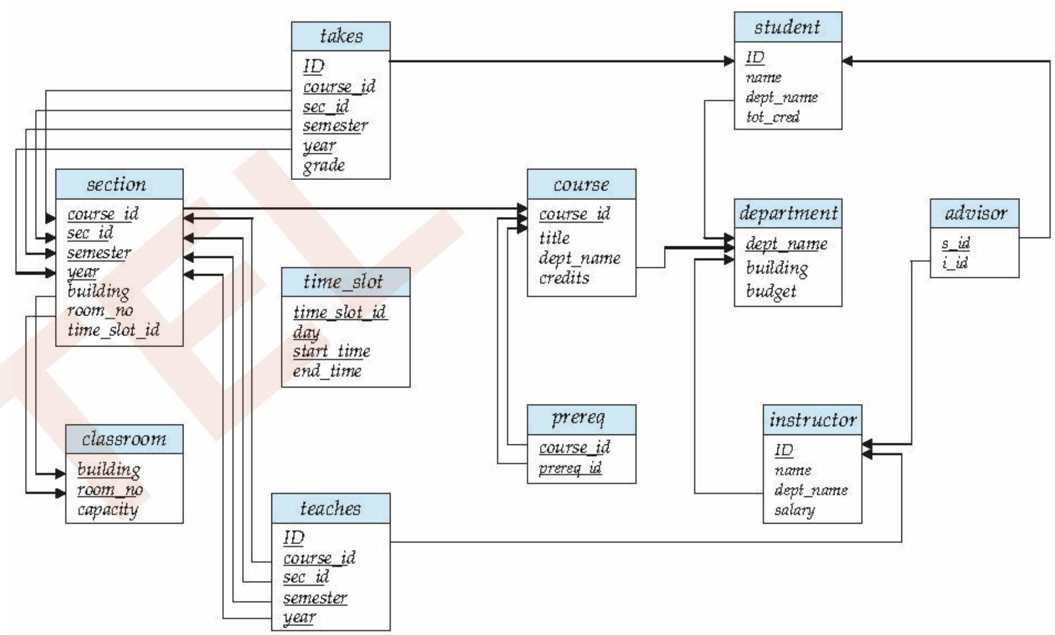



*Example:*

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

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

# And a Few More Relation Definitions

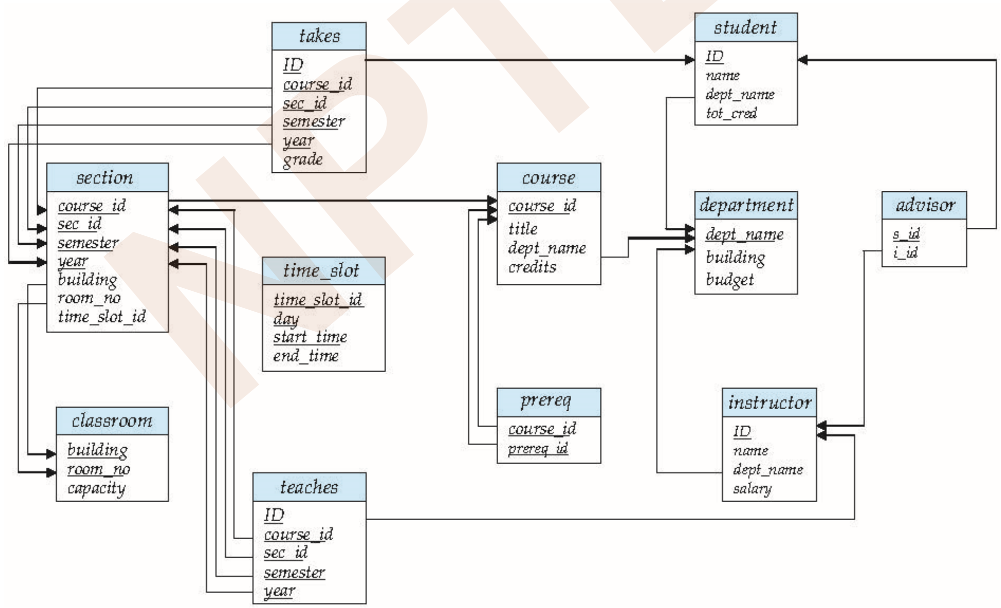



- **create table** *student* (
    *ID* **varchar**(5),
    *name* **varchar**(20) not null,
    *dept_name* **varchar**(20),
    *tot_cred* **numeric**(3,0),
    **primary key** *(ID),*
    **foreign key** *(dept_name*)
    **references** *department*);

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

# And more still

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



# Updates to tables

- **Insert**
  - **insert into** *instructor* **values** (‘10211’, ‘Smith’, ‘Biology’, 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*
    - All exiting tuples in the relation are assigned *null* as the value for the new attribute
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases

- History of SQL
- Data Definition Language (DDL)
- **Basic Query Structure (DML)**

# BASIC QUERY STRUCTURE

# 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
- 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* ≡ *NAME* ≡ *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*

# 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'

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

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

  **select** 'A'
  **from** *instructor*

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

# The select Clause (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*)
- Cartesian product 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 |
| … | … | … | … | … | … | … | … | … |
| … | … | … | … | … | … | … | … | … |

# Module Summary

- Introduced relational query language
- Familiarized with data definition and basic query structure

# Instructor and TAs

| Name | Mail | Mobile |
|---|---|---|
| Partha Pratim Das, Instructor | ppd@cse.iitkgp.ernet.in | 9830030880 |
| Srijoni Majumdar, TA | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA | himadribhuyan@gmail.com | 9438911655 |
| Gurunath Reddy M | mgurunathreddy@gmail.com | 9434137638 |



**Slides used in this presentation are borrowed from http://db-book.com/ with kind permission of the authors.**

**Edited and new slides are marked with "PPD".**

# Database Management Systems

## Module 07: Introduction to SQL/2

**Partha Pratim Das**

*Department of Computer Science and Engineering*

*Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar**
**Himadri B G S Bhuyan**
**Gurunath Reddy M**



**Database System Concepts, 6th Ed.**



**Slides marked with 'PPD' are new or edited**

# Module Recap

- History of SQL
- Data Definition Language (DDL)
- Basic Query Structure (DML)

# Module Objectives

- To complete the understanding of basic query structure and set operations
- To familiarize with null values and aggregation

# Module Outline

- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions

- **Additional Basic Operations**
- Set Operations
- Null Values
- Aggregate Functions

# ADDITIONAL BASIC OPERATIONS

# Cartesian Product

- 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*)
- Cartesian product 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 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

# Examples

- Find the names of all instructors who have taught some course and the course_id

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

  - Equi-Join, Natural Join

| *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 |

| *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~~ |

# Examples

- 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'

# 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*

# Cartesian Product Example*

- Relation *emp-super*

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

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

# 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');

# 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 $\sigma_\theta$, 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:
- $r_1 = \{(1, a)\ (2,a)\}$ $r_2 = \{(2), (3), (3)\}$
- 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** $A_1, A_2, ..., A_n$
**from** $r_1, r_2, ..., r_m$
**where** $P$

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

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

- Additional Basic Operations
- **Set Operations**
- Null Values
- Aggregate Functions

# SET OPERATIONS

# 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*

- Additional Basic Operations
- Set Operations
- **Null Values**
- Aggregate Functions

# NULL VALUES





# 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

- Three values – *true*, *false*, *unknown*
- Any comparison with *null* returns *unknown*
  - Example*: 5 < null   or   null <> null    or    null = null*
- Three-valued logic using the value *unknown*:
  - OR: (*unknown* **or** *true*)   = *true*,<br>(*unknown* **or** *false*)  = *unknown*<br>(*unknown* **or** *unknown) = unknown*
  - AND: *(true* **and** *unknown)*  = *unknown,*<br>*(false* **and** *unknown) = false,*<br>*(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*

- Additional Basic Operations
- Set Operations
- Null Values
- **Aggregate Functions**

# AGGREGATE FUNCTIONS

# 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*) **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 |

# 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*;

# 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

# 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

# Module Summary

- Completed the understanding of basic query structure and set operations
- Familiarized with null values and aggregation

# Instructor and TAs

| Name | Mail | Mobile |
|---|---|---|
| Partha Pratim Das, Instructor | ppd@cse.iitkgp.ernet.in | 9830030880 |
| Srijoni Majumdar, TA | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA | himadribhuyan@gmail.com | 9438911655 |
| Gurunath Reddy M | mgurunathreddy@gmail.com | 9434137638 |

**Slides used in this presentation are borrowed from http://db-book.com/ with kind permission of the authors.**

**Edited and new slides are marked with “PPD”.**

# Database Management Systems

## Module 08: Introduction to SQL/3



**Partha Pratim Das**

*Department of Computer Science and Engineering*
*Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar**
**Himadri B G S Bhuyan**
**Gurunath Reddy M**

**Database System Concepts, 6th Ed.**



**Slides marked with 'PPD' are new or edited**

# Module Recap

- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions

# Module Objectives

- To understand nested sub-query in SQL
- To understand processes for data modification

# Module Outline

- Nested Subqueries
- Modification of the Database

# NESTED SUB-QUERIES

# 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
- The nesting can be done in the following SQL query

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

as follows:

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

  $B$ <operation> (subquery)

  where $B$ is an attribute and <operation> to be defined later

# 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

# 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 Membership (Cont.)

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

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

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

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

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

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

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

(= **some**) $\equiv$ **in**

However, ($\ne$ **some**) $\not\equiv$ **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 <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** $\begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}$ ) = 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**) $\not\equiv$ **in**

# Test for Empty Relations

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



# 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);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

# Use of "not exists" Clause

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

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took

- *Note:* $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

- *Note:* Cannot write this query using = **all** and its variants

# 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);
```





# 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*<br>**from** (**select** *dept_name*, **avg** (*salary*) **as** *avg_salary*<br>        **from** *instructor*<br>        **group by** *dept_name*)<br>**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*<br>**from** (**select** *dept_name*, **avg** (*salary*)<br>        **from** *instructor*<br>        **group by** *dept_name*) **as** *dept_avg* (*dept_name*, *avg_salary*)<br>**where** *avg_salary* > 42000;

# With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs
- Find all departments with the maximum budget

**with** *max_budget* (*value*) **as**
    (**select max**(*budget*)
     **from** *department*)
**select** *department.name*
**from** *department*, *max_budget*
**where** *department*.*budget* = *max_budget.value*;

# Complex Queries using With Clause*

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
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
- List all departments along with the number of instructors in each department

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

- Runtime error if subquery returns more than one result tuple





- Nested Subqueries
- **Modification of the Database**



# MODIFICATION OF THE DATABASE

# Modification of the Database

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

# Deletion

- Delete all instructors

  **delete from** *instructor*

- Delete all instructors from the Finance department

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

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

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

# Deletion (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

  **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** *table*1 **select** * **from** *table*1

  would cause problem

# 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)

# 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, 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**

# Module Summary

- Introduced nested sub-query in SQL
- Introduced data modification



# Instructor and TAs

| Name | Mail | Mobile |
|---|---|---|
| Partha Pratim Das, Instructor | ppd@cse.iitkgp.ernet.in | 9830030880 |
| Srijoni Majumdar, TA | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA | himadribhuyan@gmail.com | 9438911655 |
| Gurunath Reddy M | mgurunathreddy@gmail.com | 9434137638 |



**Slides used in this presentation are borrowed from http://db-book.com/ with kind permission of the authors.**

**Edited and new slides are marked with "PPD".**

# Database Management Systems

## Module 09: Intermediate SQL/1


**Partha Pratim Das**

*Department of Computer Science and Engineering*
*Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar**
**Himadri B G S Bhuyan**
**Gurunath Reddy M**




**Database System Concepts, 6th Ed.**



**Slides marked with 'PPD' are new or edited**

# Module Recap

- Nested Subqueries
- Modification of the Database

# Module Objectives

- To learn SQL expressions for Join and Views
- To understand transactions

# Module Outline

- Join Expressions
- Views
- Transactions

- **Join Expressions**
- Views
- Transactions

# JOIN EXPRESSIONS

# Joined Relations

- **Join operations** take two relations and return as a result another relation
- 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

# Types of Join between Relations

- Cross join
- Inner join
  - Equi-join
    - Natural join
- Outer join
  - Left outer join
  - Right outer join
  - Full outer join
- Self-join

# Cross Join



- CROSS JOIN returns the Cartesian product of rows from tables in the join
  - Explicit

    **select** *

    **from** *employee* **cross join** *department*;
  - Implicit

    **select** *

    **from** *employee*, *department*;

# 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*

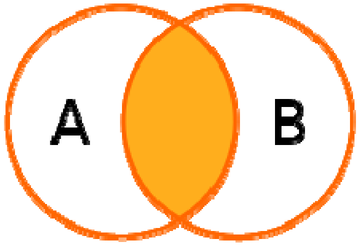
| *course_id* | *prereq_id* |
|---|---|
| BIO-301 | BIO-101 |
| CS-190 | CS-101 |
| CS-347 | CS-101 |

- Observe that

  prereq information is missing for CS-315 and

  course information is missing for CS-437

# Inner Join




- *course* **inner join** *prereq*

| *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 |

- If specified as **natural,** the 2nd course_id field is skipped

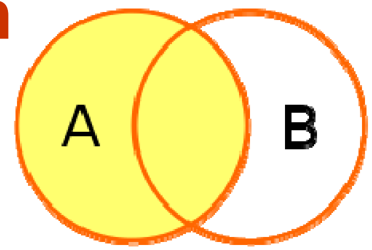
| *course_id* | *title* | *dept_name* | *credits* |
|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 |
| CS-190 | Game Design | Comp. Sci. | 4 |
| CS-315 | Robotics | Comp. Sci. | 3 |

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

# 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

# Left Outer Join



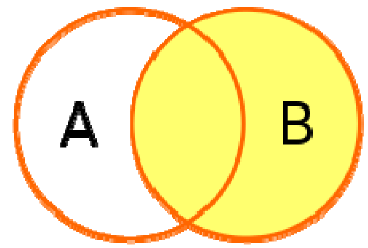

- *course* **natural left outer join** *prereq*

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

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

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

# Right Outer Join




- *course* **natural right outer join** *prereq*

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

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

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

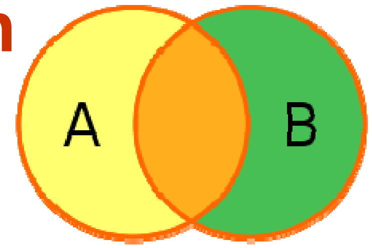
# Joined Relations

- **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** $(A_1, A_1, \ldots, A_n)$ |

# Full Outer 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 |

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

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

# Joined Relations – Examples

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

- What is the difference between the above (equi_join), and a natural join?
- *course* **left outer join** *prereq* **on** *course.course_id = prereq.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 |
| CS-315 | Robotics | Comp. Sci. | 3 | *null* | *null* |

# Joined Relations – Examples

- *course* **natural right outer join** *prereq*

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

- *course* **full outer join** *prereq* **using** (*course_id*)

| *course_id* | *title* | *dept_name* | *credits* | *prere_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 |



- Join Expressions
- **Views**
- Transactions

# VIEWS

# 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 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 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*

- Find all instructors in the Biology department

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

- Create a view of 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 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';
- **create view** *physics_fall_2009_watson* **as**
  **select** *course_id*, *room_number*
  **from** *physics_fall_2009*
  **where** *building*= 'Watson';

# View Expansion

- Expand use of a view in a query/another view

```
create view physics_fall_2009_watson as
(select course_id, room_number
from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2009')
where building= 'Watson';
```

# Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation $v_1$ is said to *depend directly* on a view relation $v_2$ if $v_2$ is used in the expression defining $v_1$
- A view relation $v_1$ is said to *depend on* view relation $v_2$ if either $v_1$ depends directly to $v_2$ or there is a path of dependencies from $v_1$ to $v_2$
- A view relation $v$ is said to be *recursive* if it depends on itself





# View Expansion*

- A way to define the meaning of views defined in terms of other views
- Let view $v_1$ be defined by an expression $e_1$ that may itself contain uses of view relations
- View expansion of an expression repeats the following replacement step:

  **repeat**
  Find any view relation $v_i$ in $e_1$
  Replace the view relation $v_i$ by the expression defining $v_i$
  **until** no more view relations are present in $e_1$
- As long as the view definitions are not recursive, this loop will terminate





# Recursive View

- In SQL, recursive queries are typically built using these components:
  - A non-recursive seed statement
  - A recursive statement
  - A connection operator
    - The only valid set connection operator in a recursive view definition is UNION ALL
  - A terminal condition to prevent infinite recursion

# Recursive View – Example

- In the context of a relation *flights*:

**create table** *flights* (
source **varchar**(40),
destination **varchar**(40),
carrier **varchar**(40),
cost **decimal**(5,0));

| source | destination | carrier | cost |
|---|---|---|---|
| Paris | Detroit | KLM | 7 |
| Paris | New York | KLM | 6 |
| Paris | Boston | American Airlines | 8 |
| New York | Chicago | American Airlines | 2 |
| Boston | Chicago | American Airlines | 6 |
| Detroit | San Jose | American Airlines | 4 |
| Chicago | San Jose | American Airlines | 2 |

- Find all the destinations that can be reached from 'Paris'

# Recursive View – Example

```
create recursive view reachable_from (source,destination,depth) as (
    select root.source, root.destination, 0 as depth
    from flights as root
    where root.source = 'Paris'
  union all
    select in1.source, out1.destination, in1.depth + 1
    from reachable_from as in1, flights as out1
    where in1.destination = out1.source and
          in1.depth <= 100);
```

- A non-recursive seed statement
- A recursive statement
- A connection operator
- A terminal condition to prevent infinite recursion

| source | destination | carrier | cost |
|---|---|---|---|
| Paris | Detroit | KLM | 7 |
| Paris | New York | KLM | 6 |
| Paris | Boston | American Airlines | 8 |
| New York | Chicago | American Airlines | 2 |
| Boston | Chicago | American Airlines | 6 |
| Detroit | San Jose | American Airlines | 4 |
| Chicago | San Jose | American Airlines | 2 |

- Get the result by simple selection on the view:

```
select distinct source, destination
  from reachable_from;
```

| source | destination |
|---|---|
| Paris | Detroit |
| Paris | New York |
| Paris | Boston |
| Paris | Chicago |
| Paris | San Jose |

This example view, *reachable_from,* is called the *transitive closure* of the *flights* relation

**Source**: https://info.teradata.com/HTMLPubs/DB_TTU_16_00/index.html#page/SQL_Reference%2FB035-1184-160K%2Fsme1472241335807.html%23wwID0EJ23T

# The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *flights* with itself
    - This can give only a fixed number of levels of reachable destinations
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of reachable destinations on which the query will not work

# The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *reachable_from*
  - The next slide shows a *flights* relation
  - Each step of the iterative process constructs an extended version of *reachable_from* from its recursive definition
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is, if we add tuples to *flights* the view *reachable_from* contains all of the tuples it contained before, plus possibly more

# Example of Fixed-Point Computation

| source | destination | carrier | cost |
|---|---|---|---|
| Paris | Detroit | KLM | 7 |
| Paris | New York | KLM | 6 |
| Paris | Boston | American Airlines | 8 |
| New York | Chicago | American Airlines | 2 |
| Boston | Chicago | American Airlines | 6 |
| Detroit | San Jose | American Airlines | 4 |
| Chicago | San Jose | American Airlines | 2 |

| Iteration # | Tuples in Closure |
|---|---|
| 0 | Detroit, New York, Boston |
| 1 | Detroit, New York, Boston, San Jose, Chicago |
| 2 | Detroit, New York, Boston, San Jose, Chicago |

# 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

# Some Updates cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  **select** *ID*, *name*, *building*
  **from** *instructor*, *department*
  **where** *instructor*.*dept_name*= *department*.*dept_name*;
- **insert into** *instructor_info* **values** ('69987', 'White', 'Taylor');
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?
- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause

# And Some Not at All

**create view** *history_instructors* **as**
  **select** *
  **from** *instructor*
  **where** *dept_name*= 'History';

- What happens if we insert ('25566', 'Brown', 'Biology', 100000) into *history_instructors?*

# Materialized Views

- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the view
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated

- Join Expressions
- Views
- **Transactions**



# TRANSACTIONS

# Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use:  **begin atomic**  ….  **end**
    - Not supported on most databases

# Module Summary

- Learnt SQL expressions for Join and Views
- Introduced transactions

# Instructor and TAs

| Name | Mail | Mobile |
|---|---|---|
| Partha Pratim Das, Instructor | ppd@cse.iitkgp.ernet.in | 9830030880 |
| Srijoni Majumdar, TA | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA | himadribhuyan@gmail.com | 9438911655 |
| Gurunath Reddy M | mgurunathreddy@gmail.com | 9434137638 |

**Slides used in this presentation are borrowed from http://db-book.com/ with kind permission of the authors.**

**Edited and new slides are marked with "PPD".**

# Database Management Systems

## Module 10: Intermediate SQL/2



**Partha Pratim Das**

*Department of Computer Science and Engineering*
*Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar**
**Himadri B G S Bhuyan**
**Gurunath Reddy M**

**Database System Concepts, 6th Ed.**



**Slides marked with 'PPD' are new or edited**

# Module Recap

- Join Expressions
- Views
- Transactions

# Module Objectives

- To learn SQL expressions for integrity constraints
- To understand more data types in SQL
- To understand authorization in SQL

# Module Outline

- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

# INTEGRITY CONSTRAINTS

# Integrity Constraints

- Integrity constraints guard against accidental damage to the database, 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





# 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**

- **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

  Example: ensure that semester is one of fall, winter, spring or summer:

  **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'))
  );

# 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 that 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

# Cascading Actions in Referential Integrity

- **create table** *course (*
  *course_id* **char**(5) **primary key**,
  *title* **varchar**(20),
  *dept_name* **varchar**(20) **references** *department*
  *)*
- **create table** *course* (
  …
  *dept_name* **varchar**(20),
  **foreign key** (*dept_name*) **references** *department*
  **on delete cascade**
  **on update cascade**,
  . . .
  )
- alternative actions to cascade: **no action**, **set null**, **set default**

# Integrity Constraint Violation During Transactions

**create table** *person* (
*ID* **char**(10),
*name* **char**(40),
*mother* **char**(10),
*father* **char**(10),
**primary key** *ID,*
**foreign key** *father* **references** *person,*
**foreign key** *mother* **references** *person*)

- How to insert a tuple without causing constraint violation ?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking (will discuss later)

- Integrity Constraints
- **SQL Data Types and Schemas**
- Authorization



# SQL DATA TYPES AND SCHEMAS

# Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** ‘2005-7-27’
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** ‘09:00:30’ **time** ‘09:00:30.75’
- **timestamp**: date plus time of day
  - Example: **timestamp** ‘2005-7-27 09:00:30.75’
- **interval:** period of time
  - Example: interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

# Index Creation

```
create table student
(ID varchar (5),
name varchar (20) not null,
dept_name varchar (20),
tot_cred numeric (3,0) default 0,
primary key (ID))
```

- **create index** *studentID_index* **on** *student*(*ID*)
- Indices are data structures used to speed up access to records with specified values for index attributes

```
select *
from  student
where  ID = '12345'
```

  - can be executed by using the index to find the required record, without looking at all records of *student*
  - *More on indices in Chapter 11*

# User-Defined Types

- **create type** construct in SQL creates user-defined type

  **create type** *Dollars* **as numeric (12,2) final**

  - **create table** *department*
    (*dept_name* **varchar** (20),
    *building* **varchar** (15),
    *budget Dollars*);

# Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  **create domain** *person_name* **char**(20) **not null**

- Types and domains are similar
- Domains can have constraints, such as **not null**, specified on them

  **create domain** *degree_level* **varchar**(10)
  **constraint** *degree_level_test*
  **check** (**value in** ('Bachelors', 'Masters', 'Doctorate'));

# Large-Object Types



- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself



- Integrity Constraints
- SQL Data Types and Schemas
- **Authorization**

# AUTHORIZATION

# Authorization

- Forms of authorization on parts of the database:
  - **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

# Authorization Specification in SQL

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

  **grant** <privilege list>

  **on** <relation name or view name> **to** <user list>

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations
- 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 $U_3$ **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
- **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 <revokee-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

- **create role** instructor;

  **grant** *instructor* **to Amit;**
- Privileges can be granted to roles:

  **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles

  **create role** *teaching_assistant*

  **grant** *teaching_assistant* **to** *instructor*;
  - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

# Authorization on Views

**create view** *geo_instructor* **as**
(**select** *
**from** *instructor*
**where** *dept_name* = ’Geology’);

**grant select on** *geo_instructor* **to** *geo_staff*

- Suppose that a *geo_staff* member issues

  **select** *
  **from** *geo_instructor*;

- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

# Other Authorization Features*

- **references** privilege to create foreign key

  **grant reference** (*dept_name*) **on** *department* **to** Mariano;

  - why is this required?
- transfer of privileges

  **grant select on** *department* **to** Amit **with grant option**;

  **revoke select on** *department* **from** Amit, Satoshi **cascade**;

  **revoke select on** *department* **from** Amit, Satoshi **restrict**;

# Module Summary

- Learnt SQL expressions for integrity constraints
- Familiarized with more data types in SQL
- Discussed authorization in SQL

# Instructor and TAs

| Name | Mail | Mobile |
|---|---|---|
| Partha Pratim Das, Instructor | ppd@cse.iitkgp.ernet.in | 9830030880 |
| Srijoni Majumdar, TA | majumdarsrijoni@gmail.com | 9674474267 |
| Himadri B G S Bhuyan, TA | himadribhuyan@gmail.com | 9438911655 |
| Gurunath Reddy M | mgurunathreddy@gmail.com | 9434137638 |

**Slides used in this presentation are borrowed from http://db-book.com/ with kind permission of the authors.**

**Edited and new slides are marked with "PPD".**