Database Systems

ECE 656
University of Waterloo

Dr. Paul A.S. Ward

Acknowledgment: slides derived from Dr. Wojciech Golab
based on materials provided by
Silberschatz, Korth, and Sudarshan, copyright 2010
(source: www.db-book.com)

Part 1: RDBMS

- What are Databases
- The Relational Model
- SQL

- Textbook reading: chapter 1, 2, 3, 4, 5.

What is the Problem?

- Data Management
  - Storage
  - Manipulation
  - Query
- Dependability
  - Against loss
  - Against corruption
- Integrity
- Transaction capacity

History of Database Systems

- 1950s and early 1960s:
  - ran on early transistor-based programmable digital computers
  - punched cards used for input (i.e., program)
  - magnetic tapes used to store data sets

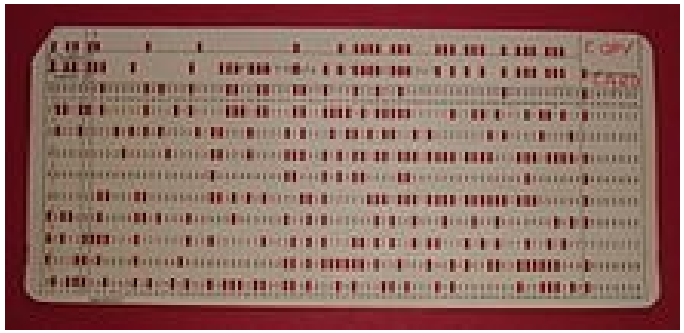


Image sources: http://en.wikipedia.org/wiki/Magnetic_tape
http://en.wikipedia.org/wiki/Punched_card

History of Database Systems

- Late 1960s and 1970s:
  - hard disks allowed more direct access to data
  - network and hierarchical data models in widespread use
  - Ted Codd (1923-2003) defines the relational data model
    - won the ACM Turing Award in 1981 for this pioneering work
    - IBM Research begins System R prototype
    - UC Berkeley begins Ingres prototype
  - high-performance (for the era) transaction processing

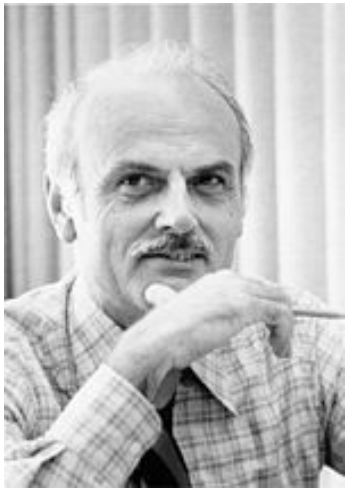


Image source:
http://en.wikipedia.org/wiki/Edgar_F._Codd

History of Database Systems

- 1980s:
  - research relational prototypes evolve into commercial systems
  - SQL becomes industry standard
  - first parallel and distributed database systems
  - object-oriented database systems
- 1990s:
  - large decision support and data-mining applications
  - large multi-terabyte data warehouses
  - emergence of Web commerce
- Early 2000s:
  - XML and XQuery standards
  - automated database administration
- Later 2000s:
  - unstructured and semi-structured data
  - scalable NoSQL systems (BigTable, PNuts, Dynamo)

Drawbacks of Using File Systems

- File systems lead to data redundancy and inconsistency.
  - multiple file formats, duplication of information in different files
- File systems cannot answer queries directly.
  - need to write a new program to carry out each new task
  - one data set may be scattered across multiple files
- File systems do not enforce integrity.
- integrity constraints (*e.g.*, account balance ≥ 0) become
  - buried in program code rather than being stated explicitly
  - adding new constraints or changing existing ones is cumbersome

Drawbacks of Using File Systems

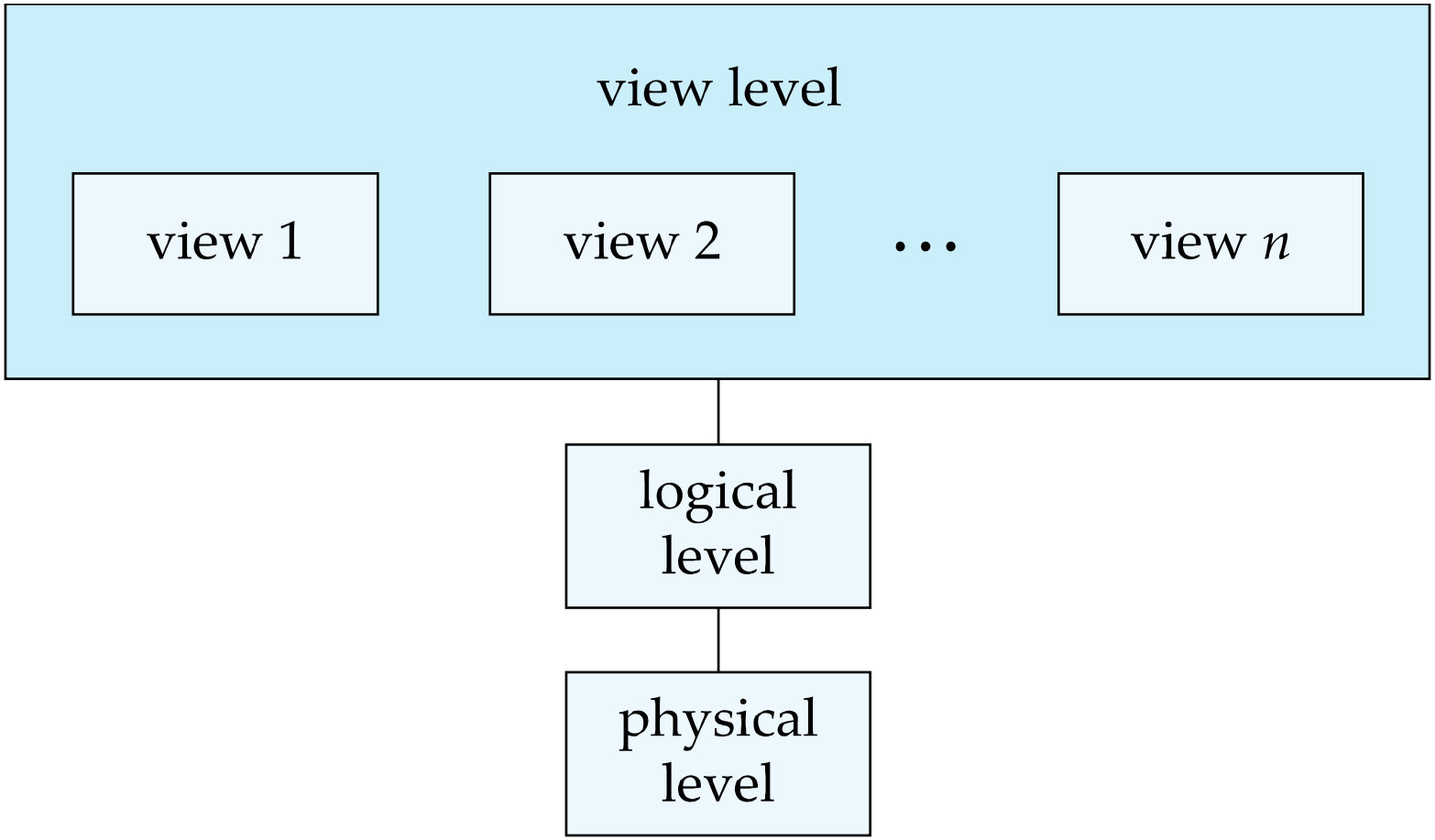
- Updates in a file system are not always atomic.
  - failures may leave data in an inconsistent state with partial updates carried out (*e.g.*, transfer of funds from one account to another)
- File systems offer limited support for concurrent access by multiple users.
  - concurrent access needed for performance
  - uncontrolled concurrent accesses can lead to inconsistencies
  - (*e.g.*, two people withdrawing money from the same account simultaneously)
- File systems do not provide sufficient security.
  - difficult to provide access to a specific subset of the data

A database management system (DBMS)
offers solutions to all the above problems!
(WTF? Be careful of statements like this. Why?)

Levels of Abstraction in a Database

- **Physical level:** describes how a record (*e.g.*, customer) is stored.
  - *e.g.*, customers are stored in ascending order by ID, and there is a secondary index on the name attribute
- **Logical level:** describes the structure of the data stored in a database, and the relationships among the data.
  - *e.g.* an instructor has an ID and name, and belongs to some department.
- **View level:** describes a virtual structure of the data imposed by the database designer on top of the logical level (e.g., a virtual table defined as the result of querying a base table or joining two base tables).
- *e.g.*, students may see an instructor's teaching evaluations, but only
  - HR staff (or the instructor) may see the instructor's salary

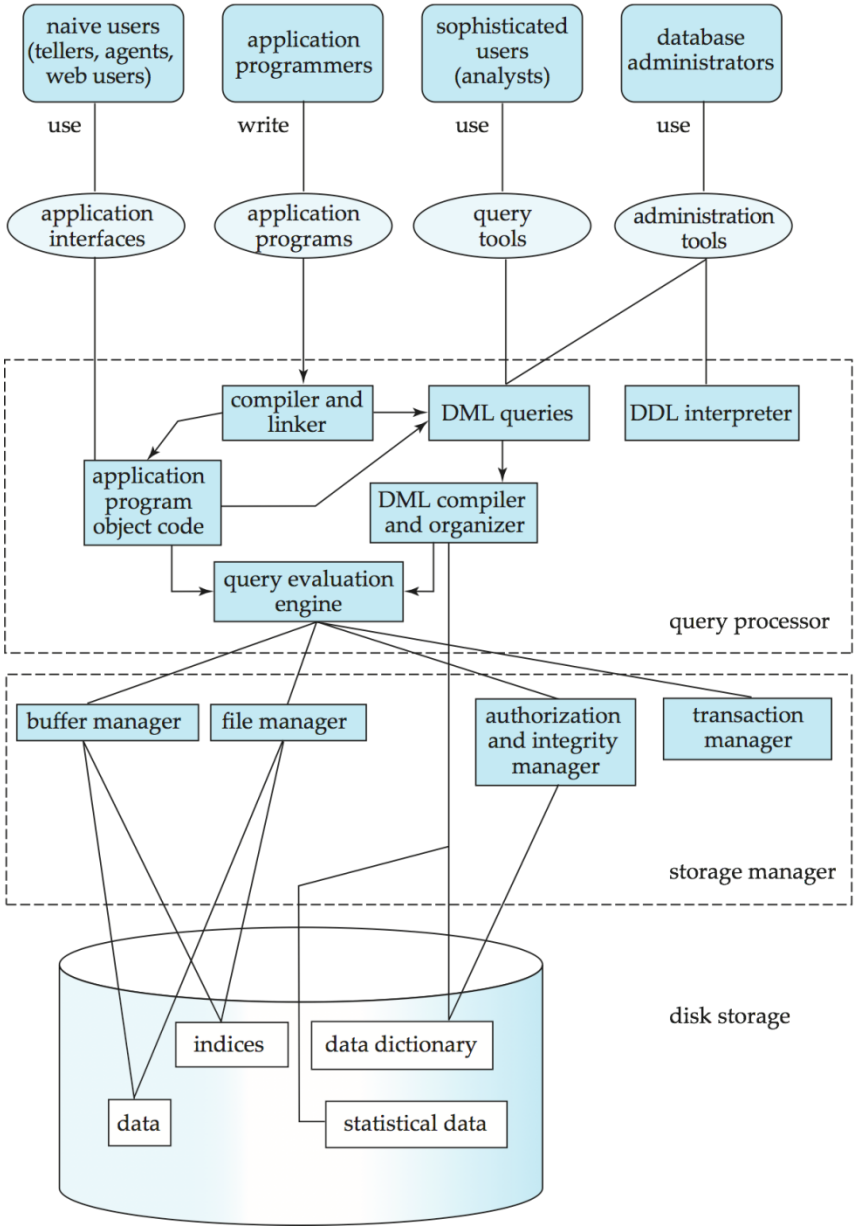
Levels of Abstraction



Physical vs. Logical Schemas

- **Schema** – the shape or structure of the database.
  - **physical schema**: design at physical level (storage, file formats, indexes)
  - **logical schema**: design at logical level (data types, relations, rows, columns)
- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema or the application program.
  - Example 1: a table that is accessed frequently can be moved to a faster disk without breaking any of the queries (and hence without breaking the applications that depend on these queries).
  - Example 2: an index can be added to speed up a specific query without breaking that query or other queries.
- **Logical Data Independence** – the ability to modify the logical schema without changing the application program.
  - Example 1: a new table can be added to the logical schema without breaking any of the queries defined over existing tables or views.
  - Example 2: a column can be added to a table without breaking any of the queries that access the table or any view defined over the table.

Structure of a Modern RDBMS



Aside: Setting up MySQL

- You should install MySQL on your own computer for convenience and as a useful practical exercise.
- The instructions in these slides cover both Linux and Windows, and were tested against the following software versions:
  - MySQL 5.5 on Ubuntu Linux 14.04 (LTS)
  - MySQL 5.6 on Windows 8.1

Setting up MySQL

Instructions for Ubuntu Linux

Step 1: provision a Linux box and obtain sudo access.

- Using an Amazon EC2 instance:
  - select Ubuntu Server 14.04 LTS (HVM) 64-bit instance
  - select instance type (*e.g.*, general purpose t2.medium)
  - edit security group by adding a custom TCP rule to allow incoming connections on port 3306 (MySQL JDBC interface) from anywhere
  - launch the instance and record its public IP address
  - access the VM using ssh as user “ubuntu” (ssh ubuntu@host)

Using your personal computer:
open TCP port 3306 in your firewall if planning remote access

- (*e.g.*, sudo ufw allow proto tcp from any to any port 3306)

Setting up MySQL

- Step 2: install and secure MySQL
  - sudo apt-get update
  - sudo apt-get upgrade
  - sudo apt-get install mysql-server
  - mysql_secure_installation
    - enter the root password you chose earlier
    - you need not change the root password (“n” to first question)
    - answer “y” to all the other questions
    - (i.e., remove anonymous users, disallow root login remotely, remove test database, reload privilege tables now)
  - If planning remote access then edit /etc/mysql/my.cnf and comment out the line with the “bind-address” property
  - sudo service mysql restart

Setting up MySQL

- Step 3: add a database, table and user
  - launch the MySQL shell: mysql -u root -p
    - enter the MySQL root password you chose earlier
  - from the mysql shell execute the following commands:
    - CREATE DATABASE myDB;
    - CREATE USER ’myUser'@'%' IDENTIFIED BY ’myPassword';
    - GRANT ALL ON myDB.* TO ’myUser'@'%';
    - EXIT;

Setting up MySQL

- Step 4: add a table
  - launch the MySQL shell again: mysql -u myUser -p
    - enter the password “myPassword”
  - from the mysql shell execute the following commands:
    - USE myDB;
    - CREATE TABLE Persons (ID INT, FirstName VARCHAR(255), LastName VARCHAR(255));

Setting up MySQL

- Step 5: execute some SQL statements using the CLI
  - SHOW TABLES;
  - SHOW GRANTS;
  - SELECT * FROM Persons;
  - INSERT INTO Persons VALUES (0, "Homer", "Simpson"), (1, "Marge", "Simpson"), (2, "Mr", "Burns");
  - SELECT * FROM Persons;
  - SELECT FirstName FROM Persons WHERE ID = 0;

Setting up MySQL

Instructions for Windows

- Step 1: provision a Windows box and obtain administrator access.
- Step 2: download and install MySQL Community Server.
  - obtain MSI installer from http://dev.mysql.com/downloads/mysql/
  - after you click the “Download” button you may be asked to log in or sign up, but note that you can bypass this step by clicking
  - “No thanks, just start my download” near the bottom of the page
  - choose “Custom” setup and select the 32-bit or 64-bit product
  - follow the installation wizard to configure the Windows firewall if planning remote access, and enter the root password
  - after the installer exits open a Windows command prompt and locate the MySQL shell
  - (*e.g.*, C:\Program Files\MySQL\MySQL Server 5.6\bin\mysql.exe)
- Repeat steps 3 and onward from the Linux instructions.

The Relational Model

- Tuples and relations
- Keys
- Schemas
- Schema diagrams

- A relation is a mathematical object, and a table is its physical embodiment.
  - $R \subseteq S_1 \times S_2 \times S_3 \ldots \times S_n$
- where each S_i is a set
  - at the base level, sets are the domain of attributes
- In an RDB, relations are tables with **rows** and **columns**. The rows may represent entities or relationships, and the columns represent attributes.

Attribute Types

- The set of allowed values for each attribute is called the **domain** of the attribute.
- Attribute values are (usually) required to be **atomic**, which means they are indivisible.
  - example: a database designer may insist that the first name and the last name be separated into distinct attributes
- The special value **null** is a member of every domain, and is used to represent missing or unknown data.
- Use null values judiciously because they lead to a number of complications. (More on this later on.)

Relation Schema and Instance

- Let A_1, A_2, …, A_n denote attributes.
- Let D_1, D_2, …. D_n denote their domains.
- $R = (A_1, A_2, \ldots, A_n)$ denotes a **relation schema** over these attributes

  Example:

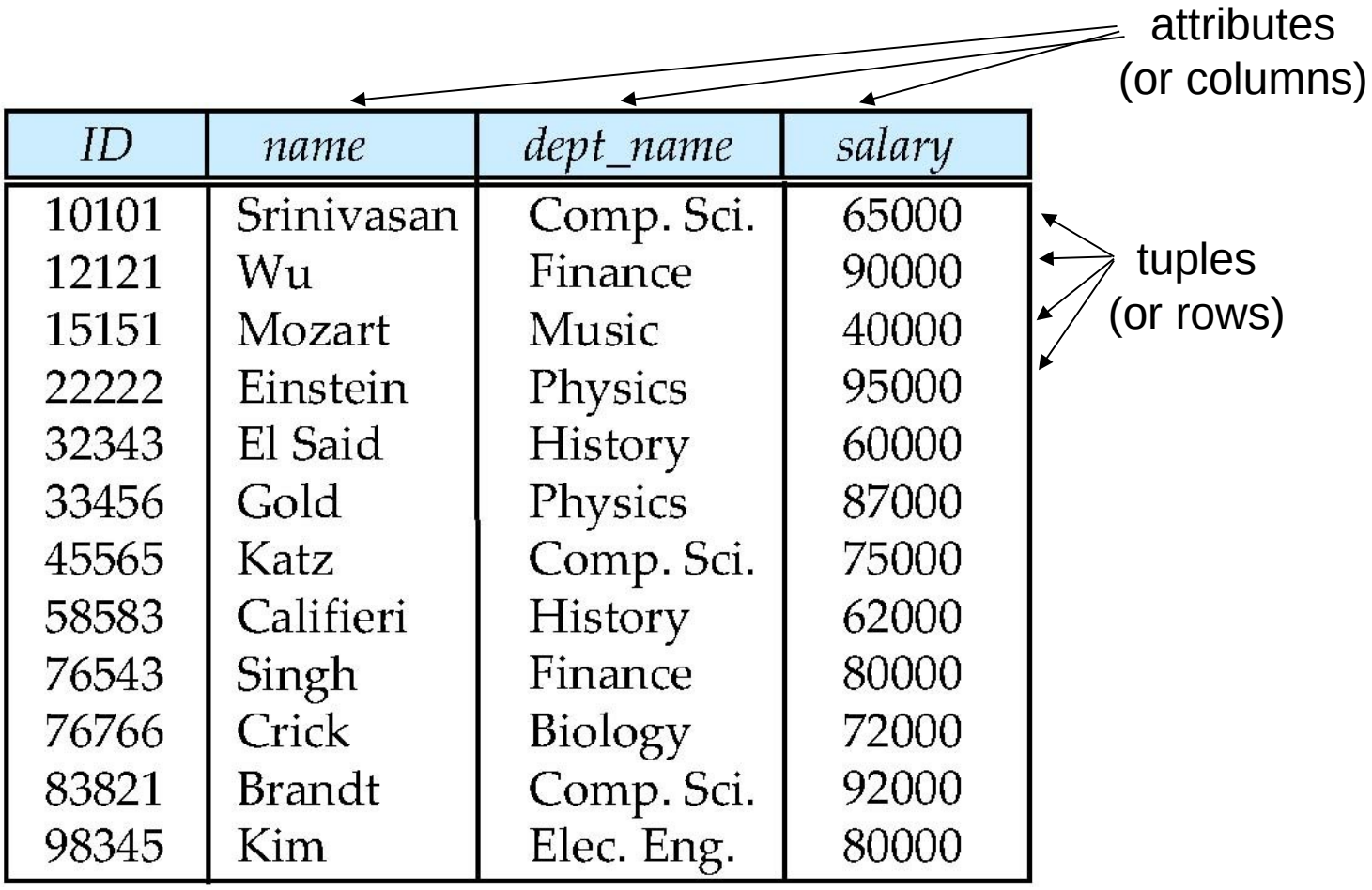
  instructor = (*ID, name, dept_name, salary*)

A **relation *r*** conforming to schema *R*, denoted as *r*(*R*), is a subset of

$D_1 \times D_2 \times \ldots \times D_n$

- Thus, a relation is a set of *n*-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$.
- An element ***t*** of ***r*** is a **tuple** (specifically an **n-tuple**), and
- corresponds to a row in a table.
- Note: the order of elements in the tuple does not matter as long as we remember the attribute corresponding to each tuple element.
- A **relation instance** refers to the concrete values of a relation.
- (*E.g.*, set of Waterloo instructors as of 10am on January 9, 2014.)

Example of a Relation Instance



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

Relations are Unordered

- Order of tuples is irrelevant (tuples may be listed in an arbitrary order).
- Example: *instructor* relation with tuples ordered arbitrarily.

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

Database

- A database typically comprises many relations.
- In the design process, information about an enterprise is broken up:
- *instructor*
- *student*

advisor

Sometimes, database designers make questionable decisions:

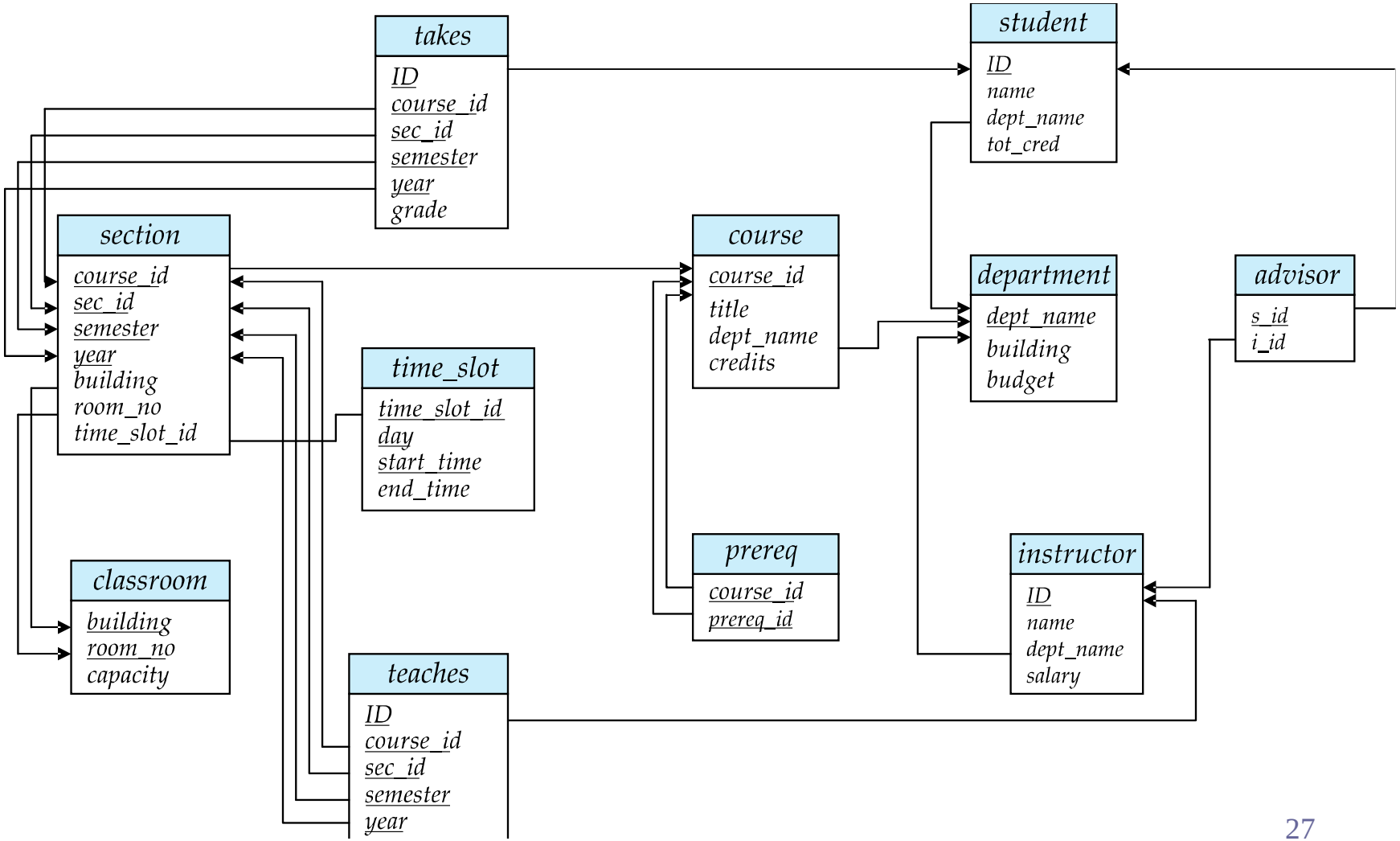
- *univ* (*instructor_ID, name, dept_name, salary, student_ID*, ..)
  - repetition of information (*e.g.*, two students have the same instructor)
  - the need for null values (*e.g.*, represent a student with no advisor)
- Normalization theory (covered later in the course) deals with how to design good relational schemas that satisfy a very precise notion of “goodness”.

Keys

- Let *R* be a relation schema and let $K \subseteq R$ (*K* is a subset of *R*'s attributes).
- **Superkey** and **candidate key** are defined as in the E-R model:
  - *K* is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
    - Example: {*ID*} and {*ID*, *name*} are both superkeys of *instructor.*
    - Superkey *K* is a **candidate key** if *K* is minimal
  - Example: {*ID*} is a candidate key for *instructor*
  - One of the candidate keys is selected to be the **primary key**.
  - (Which one?)
- **Foreign key** constraint: an attribute value in one relation that must appear in another relation.
  - **referencing relation** contains a **foreign key**
  - **referenced relation** contains a **referenced key**
  - (usually the primary key)

Relational Schema Diagram for University DB

Example: Attribute *ID* in *takes* (referencing relation) is a foreign key that references *ID* in *student* (referenced relation).



SQL Basics

- DML syntax and semantics
  - Selection, projection, Cartesian product and joins, renaming
  - set operations
  - aggregation and grouping
  - ordering results
  - nested subqueries
- DDL syntax and semantics
  - SQL data types
  - creating tables, constraints, views
  - insert, update, and delete rows
  - auto-increment attributes
  - stored procedures, cursors, and triggers

Brief History of SQL

- SEQUEL (Structured English Query Language) developed for *System R* project at IBM San Jose Research Lab in early 1970s. Later shortened to SQL (Structured Query Language).
- Standardized by ANSI (since 1986) and ISO (since 1987):
  - SQL-86 (first cut)
  - SQL-89 (added integrity constraints)
  - **SQL-92** (DATE and VARCHAR types, NATURAL JOIN)
  - SQL:1999 (regular expressions, recursive queries, triggers)
  - SQL:2003 (XML support, auto-generated values)
  - SQL:2008 (revisions to cursors and triggers)
  - SQL:2011 (support for temporal databases)
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

SQL DDL, DML, DCL, TCL

- The SQL **Data Definition Language (DDL)** provides the ability to specify schema includes domains and integrity constraints.
  - *e.g.*, creating tables and declaring foreign keys
- The SQL **Data Manipulation Language (DML)** provides the ability to query data, as well as insert/delete/update tuples.
  - *e.g.*, SELECT statement
- The SQL **Data Control Language (DCL)** provides the ability to grant access privileges to users and to revoke such privileges.
- The SQL **Transaction Control Language (TCL)** provides the ability to control the execution of transactions.
- We will cover **SQL DML** and **DDL**.

Schema for Running Example

- *instructor* (*ID*, *name*, *dept_name*, *salary*)
- *teaches* (*ID*, *course_id*, *sec_id*, *semester*, *year*)
- *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_no*, *time_slot_id*)
- *course* (*course_id*, *title*, *dept_name*, *credits*)

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

Basic Query Structure

- A simple SQL query has the form:

select $A_1, A_2, ..., A_n$ ← Not all attributes need be selected
from $t_1, t_2, ..., t_m$ ← Tables to be joined
where P ← A condition over the table join

  - A_i represents an attribute
  - t_i represents a table
  - P is a predicate over the joined tables

- The result of an SQL query is a multi-set relation.

What is $t_1, t_2, \ldots, t_m$?

- The most basic form of joining two tables t_1 and t_2 is selecting all rows from t_2 and appending them to each row of t_1.
- You may think of this as the following:

```
for each row r1 in t1
    for each row r2 in t2
        output r1, r2
```

- There are better ways of joining tables than this, so do not think of this as ideal.

The select Clause

- The **select** clause lists the requested attributes in the result of a query.
- Example: find the names of all instructors:

select *name*
from *instructor*

- In general relation names are case-sensitive but attribute names are not.
  - (Some exceptions, like MySQL running on Windows.)

The select Clause (Cont.)

- The keyword **distinct** eliminates duplicates. The keyword **all** keeps duplicates. In MySQL, the default is **all**.
- Example: find the names of all departments with at least one instructor, and remove duplicates.

  select distinct *dept_name*
  from *instructor*

- Example: find the names of all departments with at least one instructor, and keep duplicates.

  select all *dept_name*
  from *instructor*

The select Clause (Cont.)

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

  select * **from** *instructor*

- The **select** clause can contain arithmetic expressions involving the operators +, –, ∗, 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 (and renamed “salary/12”).

The where Clause

- The **where** clause specifies conditions that the result must satisfy.
- Example: find all instructors in Physics with *salary* > 80000.

  select *name*
  from *instructor*
  where *dept_name* = 'Physics' **and** *salary* > 80000
- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

String Pattern Matching

- SQL includes a string matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters (wildcards):
  - percent (%): matches any substring
  - underscore (_): matches any one character
  - Example: Find the names of all instructors whose name includes the substring “ar”.

    select *name*
    from *instructor*
    where *name* **like** '%ar%'
- Example: Match the string “100 %”.

  like '100 \%' **escape** '\'
- Note: Patterns are case-sensitive.

The from Clause

The **from** clause lists the relations involved in the query.

- Example: find the Cartesian product *instructor* x *teaches*:

  select *
  from *instructor, teaches*

- This query generates every possible instructor-teaches pair,
- with all attributes from both relations.

Warning: the result set of a Cartesian product can be very large!

Cartesian Product: *instructor X teaches*

instructor

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

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

Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common attribute.
- Example: **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

Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name that get equated incorrectly.
- Example: list the names of instructors along with the titles of courses that they teach.
  - Incorrect solution (makes course.dept_name = instructor.dept_name):
    select *name*, *title*
    from *instructor* **natural join** *teaches* **natural join** *course*
  - Correct solution:
    select *name*, *title*
    from *instructor* **natural join** *teaches*, *course*
    where *teaches.course_id* = *course.course_id*
  - Another correct solution:
    select *name*, *title*
    from (*instructor* **natural join** *teaches*)
    inner join *course* **using** (*course_id*)

More Joins

- **Join operations** take two relations and return another relation as a result.
- **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 $(A_1, A_1, \ldots, A_n)$

More Joins (Cont.)

- 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 and Outer Joins

select * **from** *course* **inner join** *prereq* **on**
course.course_id = prereq.course_id

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

Note: some databases allow you to omit the keyword **"inner"**.

select * **from** *course* **left outer join** *prereq* **on**
course.course_id = prereq.course_id

course_id	*title*	*dept_name*	*credits*	*prereq_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*

Outer Joins

- **select** * **from** *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

select * **from** *course* **full outer join** *prereq* **using** (*course_id*)

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

- Note: full outer join is not supported in MySQL 5.0.

Theta Join

- A Theta join is obtained using an **inner join** with the **on** clause.
- Example: find the course ID, semester, year and title of each course offered in the Physics department.

select *section.course_id, semester, year, title*
from *section* **inner join** *course*
on (*section.course_id* = *course.course_id* **and**
dept_name = 'Physics')

The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:
  old-name **as** *new-name*
- Example:
  select *ID, name, salary/12* **as** *monthly_salary*
  from *instructor*

- Find the names of all instructors who have a higher salary than some instructor in 'Physics'.

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

  Keyword **as** is optional and may be omitted
  instructor **as** *T* ≡ *instructor T*

- (Note: keyword **as** must be omitted in Oracle)

Set Operations

- Find courses offered in Fall 2009 or in Spring 2010

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

- Find courses offered in Fall 2009 and in Spring 2010

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

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

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

Note: MySQL does not support **intersect** or **except.** Remedy?

Set Operations

- Set operations **union**, **intersect**, and **except** automatically eliminate 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*

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

Use the keyword **distinct** to eliminate duplicates.

- Example: ... **count**(**distinct** *ID*) ...

Aggregate Functions (Cont.)

- Example: find the average salary of instructors in the Physics department.

  select avg(*salary*)
  from *instructor*
  where *dept_name*= 'Physics'

- Example: 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

- Example: find the number of tuples in the *course* relation.

  select count(*)
  from *course*

- Note: avoid spaces between the aggregate function and (

Aggregation with Grouping

- Find the average salary of instructors in each department

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

- Note: departments with no instructor will not appear in result

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

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

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in the **group by** list:
- The following has undefined results:

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

Returns a group for each distinct *dept_name* in relation *instructor*.

Note: MySQL 5.5 does accept the above syntax. For each department group it will return one instructor name from that group.

Aggregation with “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.

Ordering the Display of Tuples

Example: list in alphabetical order the names of all instructors.

select distinct *name*
from *instructor*
order by *name*

- Specify **desc** for descending order or **asc** for ascending order, for each attribute. Ascending order is the default.
  - example: **order by** *name* **desc**
- Possible to sort on multiple attributes.
- example:

  select distinct *dept_name, name*
  from *instructor*
  order by *dept_name* **asc**, *name* **desc**

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.
- Common uses of subqueries including performing tests for set membership (*e.g.*, a value is in the set or not) and set cardinality (*e.g*., set is empty or not), as well as performing set operations (*e.g*., union, intersect, difference).
- The **in** construct is used for testing set membership. It returns **true** if the value on the left is an element of the relation generated by a subquery on the right.

select … **from** … **where** *value* **in** (*subquery*)
select … **from** … **where** *value* **not in** (*subquery*)

Testing for Set Membership

- Example: 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)
```

- Example: 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)
```

Testing for Empty Relations

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

Correlated subquery: uses values from outer query in **where** clause.

Test for Empty Relations (Cont.)

- Example: 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 as C
                    where C.dept_name = 'Biology' and
                    not exists
                     (select T.course_id from takes as T
                      where C.course_id = T.course_id
                      and S.ID = T.ID) ) )
```

Scalar Subquery

- **Scalar subquery** is one that yields a single value.
- Example:

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

Example:

```
select name
from instructor
where  salary * 10 >
      (select budget  from department
       where department.dept_name = instructor.dept_name)
```

- Note: An error occurs if the subquery returns more than one result tuple.

Modifying Relations – Insertion

- Example: Add a new tuple to *course.*

  insert into *course*
  values ('ECE-656', 'Databases', 'ECE', 0.5)

- Example: Add a new tuple to *course* with *credits* set to *null*.

  insert into *course*
  values ('ECE-656', 'Databases', 'ECE', *null*)

Modifying Relations – Updates

- Example: Give a 3% salary increase to all instructors whose salary is below $80,000.

```
update instructor
   set salary = salary * 1.03
   where salary < 80000
```

Modifying Relations – Deletion

- Example: Delete all instructors.

  delete from *instructor*

Example: Delete all instructors from the Math department.

delete from *instructor*
where *dept_name*= 'Math'

- Example: Delete all instructors whose departments are in the EIT building.

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

Data Definition Language

The SQL **data-definition language (DDL)** is used to specify of information about relations, including:

- the schema for each relation
- the domain of values associated with each attribute
- integrity constraints
- the set of indexes to be maintained for each relation
- the physical storage structure of each relation
- (*e.g.*, choice of InnoDB vs. MyISAM storage engine in MySQL)

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character string, with user-specified maximum length *n.*
- **int.** Integer (a machine-dependent finite subset of the integers).
- **smallint.** Small integer (a machine-dependent subset of int).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* significant digits, with *d* 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.
- **date, time**. Calendar date (YYYY-MM-DD format), and time of day (hh:mm:ss format).

Create Table Construct

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

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

- [Note: *r* is a relation, A_i is an attribute, and D_i is the domain of A_i]

- Example:

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

Views

- In some cases, it is not desirable for all users to see all the relations stored in a database instance.
- Consider a person who needs to know an instructor's name and department, but not the salary. This person should see a relation described in SQL by the following query:

  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 follows the form

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

  where <query expression> is any legal SQL query.
- Once a view is defined, the view name can be used to refer to the virtual relation.
- A view need not be defined over a single table. It can be defined over the result set of a join or group by query.
- Note: Views are dynamic. In other words, changing the data in the relations referenced by a view causes analogous changes in the virtual relation corresponding to 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*

select * **from** *departments_total_salary*

Drop and Alter Table Constructs

- **drop table** *student*
  - Deletes the table and its contents. Assumes table exists.
- **drop table if exists** *student*
  - Deletes the table and its contents if table exists.
- **delete from** *student*
  - Deletes all tuples from table, but retains table.
- **alter table**
  - **alter table** *r* **add** *A D*
    - Adds attribute with name *A* and domain *D* to relation *r.* All existing tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - Drops attribute *A* from relation *r*. Not supported by many databases.

Declaring Integrity Constraints

- **not null:** disallows null values
- **primary key** (A_1, ..., A_n): ensures uniqueness
- **unique** (A_1, ..., A_n): ensures uniqueness (think superkey)
- **foreign key** (A_m, ..., A_n) **references** *r* (A'_m, ..., A'_n):
- defines a foreign key in the child (referencing) table that points to a referenced key in a parent (referenced) table *r*
- **default** *V*: makes *V* the default value for an attribute
- Example:

```
create table instructor (
        ID              char(5),
        name            varchar(20)     not null,
        dept_name       varchar(20),
        salary          numeric(8,2)    default 0,
        primary key (ID),
        foreign key (dept_name) references department (dept_name)
    )
```

Primary Keys and Superkeys

Example of **primary key** and **unique** constraints in SQL:

```
create table customer (
        customer_id             int,
        social_insurance_num    numeric(9,0),
        first_name              varchar(20),
        last_name               varchar(20),
        primary key (customer_id),
        unique (social_insurance_num)
);
```

- Note 1: **primary key** and **unique** both identify superkeys.
- You can use **primary key** at most once per table but you can use **unique** more than once.
- Note 2: A primary key attribute cannot be null. The **primary key** constraint on an attribute implies the **not null** constraint.
- Note 3: For a **unique** attribute, the number of tuples that may have a *null* value is system-dependent (zero, one, or many).

Foreign Keys

Example of a **foreign key** (a.k.a. **referential integrity**) constraint:

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

create table department (
        dept_name    varchar(20)  primary key,
        building  varchar(20)  not null,
        budget    int
       );
```

- Note: In MySQL the referenced key (in this case *dept_name*) must be a superkey of the referenced table, or a prefix of a multi-attribute primary key of the referenced table.

Referential Actions

- **Referential actions** define the behavior of the DB in cases when an update or deletion on the parent (*i.e.*, referenced) table SQL statement affects a value referenced by a child (*i.e.*, referencing) table.
- SQL 92 defines four actions:
  - **cascade**: automatically update or delete foreign key in matching rows of child table.
  - **set null**: set foreign key columns to *null* in matching rows of child table. (Assumes foreign key is nullable.)
  - **set default**: set foreign key columns to the default value in matching rows of child table.
  - **no action**: reject operation and generate error.
  - (Also known as **restrict** in MySQL.)

Referential Actions (Cont.)

Example: cascade on update, set null on delete

```
create table instructor (
        ID          char(5)  primary key,
        name        varchar(20) not null,
        dept_name   varchar(20),
        salary      numeric(8,2),
        foreign key (dept_name) references department(dept_name)
        on update cascade  on delete set null
);
```

Foreign Key Caveats in MySQL 5.0

- MySQL will enforce foreign key constraints only in some cases:
  - The storage engine for the parent and child tables must support foreign keys (*e.g.*, InnoDB).
  - Corresponding attributes in parent and child table must have “similar internal data types”.
  - (*e.g.*, integers of the same size and sign, or strings of possibly different lengths.)
  - Foreign key and referenced key must be indexed.
- Note: we will discuss storage engines and indexing later on in the course.
- For further information consult the MySQL reference manual:
- http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html

Check Constraints

In SQL 92, constraints can be expressed using predicates and declared using the **check** clause.

- (Not supported in MySQL 5.0.)
- Example: ensure that salaries observe institutional limits.

```
create table instructor (
        ID          char(5)             primary key,
        name        varchar(20)         not null,
        dept_name   varchar(20),
        salary      numeric(8,2),
    foreign key (dept_name) references department(dept_name),
        check (salary > 50000 and salary < 150000 )
);
```

Auto-increment Attributes

- Auto-increment attributes can be used to automatically generate primary key values.
- Example:

```
create table instructor_auto (
        ID          int                 auto increment primary key,
        name        varchar(20),
        dept_name   varchar(20),
        salary      numeric(8,2)
)

insert into instructor_auto (name, dept_name, salary)
values ('Watson', 'Biology', 90210)

select * from instructor_auto
```

- Note 1: *ID* = 1 should be assigned automatically to the first tuple.
- Note 2: The auto-increment type does not exist in the ER model and therefore it does not eliminate the need for weak entity sets.

Stored Procedures

- A **stored procedure** is a subroutine that consolidates and centralizes logic that would otherwise be implemented in database applications. Using stored procedures reduces redundant SQL code in applications.
- Example:

```
DELIMITER @@
CREATE PROCEDURE ProcTopSalary
(IN dept_name VARCHAR(20))
BEGIN
      SELECT max(salary) FROM instructor
      WHERE instructor.dept_name = dept_name;
END @@
DELIMITER;
CALL ProcTopSalary('Physics');
```

Stored procedures are generally long-lived, similarly to tables.
A stored procedure can be deleted using the drop keyword:

```
DROP PROCEDURE ProcTopSalary;
```

Cursors

- A database **cursor** is a control structure that enables traversal of records in a database. Cursors are similar to iterators in Java, and are used inside stored procedures.
- Example: assign every instructor in Biology department to teach 'BIO-101', one instructor per year, consecutive years starting at 2016.
-

```
DELIMITER @@
CREATE PROCEDURE CursorDemo()
BEGIN
    DECLARE year INT DEFAULT 2016;
    DECLARE my_id VARCHAR(5);
    DECLARE my_dept_name VARCHAR(20);
    DECLARE done INT DEFAULT 0;
    DECLARE cur CURSOR FOR
        SELECT ID, dept_name from instructor;
    DECLARE CONTINUE HANDLER FOR
        NOT FOUND SET done = 1;
```

< continued on next slide >

Cursors (Cont.)

```
OPEN cur;
my_loop: LOOP
        FETCH cur INTO my_id, my_dept_name;
        IF done = 1 THEN
                LEAVE my_loop;
        END IF;
        IF my_dept_name = 'Biology' THEN
                INSERT INTO teaches
                VALUES (my_id, 'BIO-101', 1, 'Summer', year);
        END IF;
        SET year = year + 1;
END LOOP my_loop;
CLOSE cur;
END @@
DELIMITER;

CALL CursorDemo();
```

Triggers

- A **trigger** is procedural code that is automatically executed in response to certain events on a particular table or view in a database.
- MySQL supports three types of trigger events: insert, update, and delete. A trigger can be executed either before or after an event.
- Example of using triggers for validation:

```
DELIMITER @@
CREATE TRIGGER SalaryTrigger
BEFORE UPDATE ON instructor FOR EACH ROW
BEGIN
    IF NEW.salary > OLD.salary * 1.10 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'increase higher than 10%';
    END IF;
END; @@
DELIMITER;

UPDATE instructor SET salary = salary * 3.14
WHERE salary < 80000;
```