

Database Management Systems

DBMS IT214

(3-0-3-4.5)

Rachit Chhaya

rachit_chhaya@daiict.ac.in

FB3 , Room no. 3109

Overview

- **Course Handout**
 - Resources
 - Evaluation Scheme
- **Course Plan**
 - Course Overview

Resources

- **Text Book**
 - Silberschatz, Korth & Sudarshan, *Database System Concepts*, 7th Edition, McGraw-Hill, 2019
- **Additional Books**
 - Ramakrishnan & Gehrke, *Database Management Systems*, McGraw-Hill, 2003
 - Elmasri & Navathe, *Fundamentals of Database Systems*, Pearson Education 1999
 - Date C.J., *An Introduction to Database Systems*, Addison-Wesley, 2001
- **Lecture folder**

Operational Details

- **Class**
 - (Tue, Thu)10:00 am, Fri 11:00 am @ CEP 110
- **Lab Work**
 - 3 hours/ week
 - Each Lab will be evaluated. Student must attend and complete each lab in order to even pass the course
- **Exams**
 - MidSem Exam/s
 - EndSem Exam

Evaluation Scheme

- Labs & Assignments 30%
- MidSem Exam/s 30%
- EndSem Exam 40%

Database and DBMS

- **Database System DBS**
 - Collection of interrelated data, describing activities of one or more related organizations.
 - **Example:** DAIICT database contains information about students, faculty, courses, classrooms, exams
 - Set of programs to access those data
- Database Management System DBMS is a software designed to assist in maintaining and utilizing large collections of data
- Alternative is to store the data in files and use application specific codes

Data Management

- Defining structures for data storage
- Providing mechanism for accessing this information

Why Study Databases?

- Shift from computation to information
- Volume, velocity, and variety of data
- Digital libraries, biological databases

Databases Everywhere!!!

- Database Applications
 - Banking: all transactions
 - Airlines: reservations, schedules
 - Universities: registration, grades
 - Sales: customers, products, purchases
 - Online retailers: order tracking, customized recommendations
 - Manufacturing: production, inventory, orders, supply chain
 - Human resources: employee records, salaries, tax deductions

Databases touch all aspects of our lives

DBMS Functionalities

- Define a database : in terms of data types, structures and constraints
- Construct or Load the Database on a secondary storage medium
- Manipulating the database : querying, generating reports, insertions, deletions and modifications to its content
- Concurrent Processing and Sharing by a set of users and programs – yet, keeping all data valid and consistent
- Crash Recovery

The DBMS Marketplace

- Relational DBMS companies – Oracle, Sybase – are among the largest software companies in the world.
- IBM offers its relational DB2 system. With IMS, a non relational (hierarchical) system, IBM is by some accounts the largest DBMS vendor in the world.
- Microsoft offers SQL-Server, plus Microsoft Access for the cheap DBMS on the desktop
- Relational companies also challenged by object-oriented DB companies.
- But countered with object-relational systems, which retain the relational core while allowing type extension as in OO systems.

Studying DBMS

(1) Modeling and design of databases

 Data, Structuring the data

(2) Programming

 queries and DB operations like update

 SQL (DDL, DML) Data definition language, Data Manipulation Language

(3) DBMS implementation

IT214 = (1) + (2), while (3) is to be covered partly

Course Plan

ER model stands for an Entity-Relationship model.

- **Database Overview**
Basic Definitions, Data Storage, Data Models, Queries, Query Optimization, Transaction Management, Distributed Databases
- **What Data?** Requirements Collection and Analysis
- **Structuring Data (Data models)**
 - E-R Model, Conceptual Design using E-R Model
 - Relational Model
 - introduction, database architecture, integrity constraints, database design
- **Query Languages:** Relational Algebra, SQL
- **Conceptual Database Design & Tuning**
 - relational data model
 - FD, Normal Forms, Decomposition, Normalization, Schema Refinement

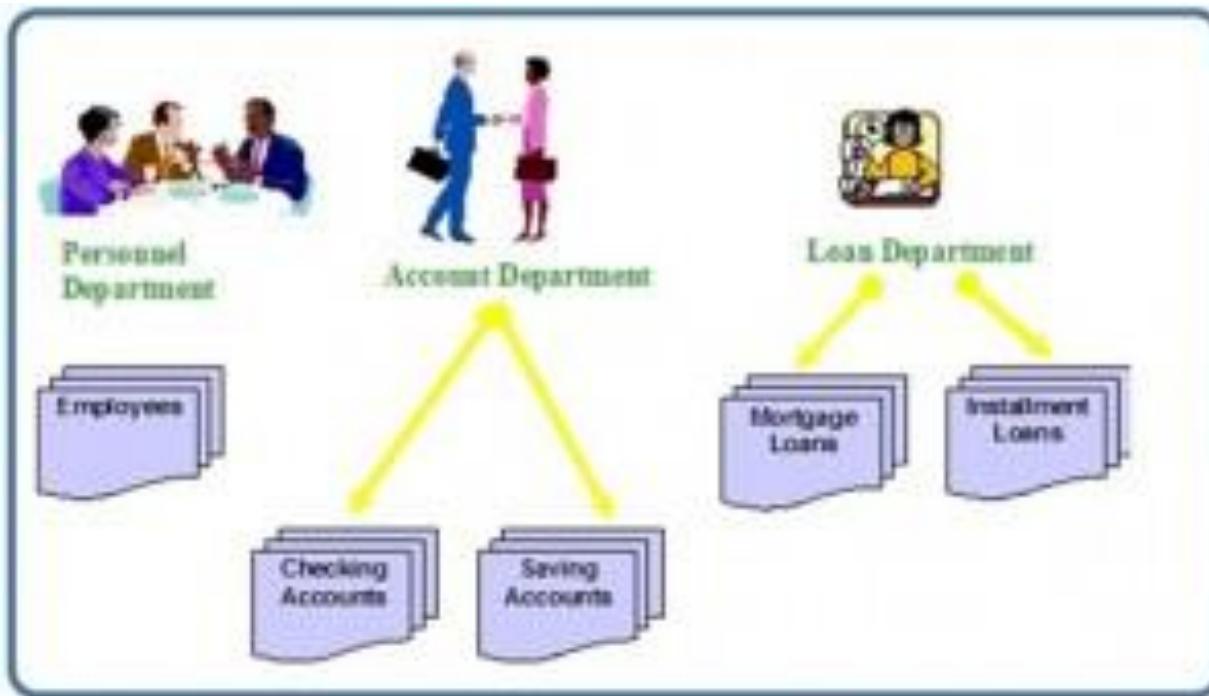
Course Plan

- **Data Storage and Indexing**
 - Storage/organization of data on disks, tapes
 - Indexing
- **Query Processing and Optimization**
 - Query Cost, Evaluation Plans, Materialized Views
- **Parallel and Distributed Databases**
 - Architecture, Storage, Query Processing
- **Transaction Management**
 - ACID Properties, Concurrency Control, Crash Recovery
- **Issues in Modern Databases**
- **Database Administration and DBM Tools**

Oracle, [postgresql/ postgres](#)

Purpose and Nature of Database Systems

File System Example



Early Days Database Applications

- **Data redundancy and inconsistency**
 - data is stored in multiple file formats resulting in duplication of information in different files
- **Difficulty in accessing data**
 - Need to write a new program to carry out each new task
- **Data isolation**
 - Multiple files and formats
- **Integrity problems**
 - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
 - Hard to add new constraints or change existing ones

Early Days Database Applications

- **Atomicity of updates**
 - Failures may leave database in an inconsistent state with partial updates carried out
 - Example: Transfer of funds from one account to another should either complete or not happen at all
- **Concurrent access by multiple users**
 - Concurrent access needed for performance
 - Uncontrolled concurrent accesses can lead to inconsistencies
 - Ex: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- **Security problems**
 - Hard to provide user access to some, but not all, data

Database Systems

- **Database systems offer solutions to these problems**
 - Data redundancy & inconsistency
 - Data access
 - Data isolation
 - Integrity problems
 - Atomicity of updates
 - Concurrent accesses by multiple users
 - Security issues

SeAtCon RedIsoAcc

What is a DBMS?

- A very large integrated collection of data
- Models real world entities
- Database Management System DBMS is software designed to assist in maintaining and utilizing large collections of data

Advantages of a DBMS

- Program-Data Independence
 - Insulation between programs and data: Allows changing data storage structures and operations without having to change the DBMS access programs.
- Efficient Data Access
 - DBMS uses a variety of techniques to store & retrieve data efficiently
- Data Integrity & Security
 - Before inserting salary of an employee, the DBMS can check that the dept. budget is not exceeded
 - Enforces access controls that govern what data is visible to different classes of users

Advantages of a DBMS

- **Data Administration**
 - When several users share data , centralizing the administration offers significant improvement
- **Concurrent Access & Crash Recovery**
 - DBMS schedules concurrent access to the data in such a manner that users think of the data as being accessed by only one user at a time
 - DBMS protects users from the ill-effects of system failures
- **Reduced Application Development Time**
 - Many important tasks are handled by the DBMS

Data Models

Structuring the Data

Data Models

- A data model is collection of concepts for describing data
- A schema is a description of a particular collection of data, using the given data model
- A relational data model is the most widely used data model today
- Network Model, Hierarchical Model, Object-Oriented Model, Relational Model, Knowledge Bases , XML, RDF, OWL, Graph Databases

Data Models

- Relational Model (E.F.Codd 1970)
 - Relation is a table with rows and columns
 - Every relation has a schema that describes the columns or fields
- Relational model is good for:
 - Large amounts of data, simple operations
 - Navigate among small number of relations
 - Difficult Applications for relational model
 - VLSI Design (CAD in general)
 - CASE
 - Graphical Data

Data Models

Where number of relations is large, relationships are complex

- Object Data Model

Object Data Model

- Complex Objects – Nested Structure (pointers or references)
- Encapsulation, set of Methods/Access functions
- Object Identity
- Inheritance – Defining new classes like old classes

Relational Model

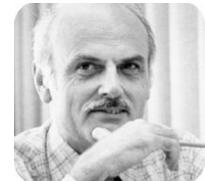
- All the data is stored in various tables.
- Example of tabular data in the relational model

Columns

Rows

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

(a) The *instructor* table



Ted Codd
Turing Award 1981

A Sample Relational Database

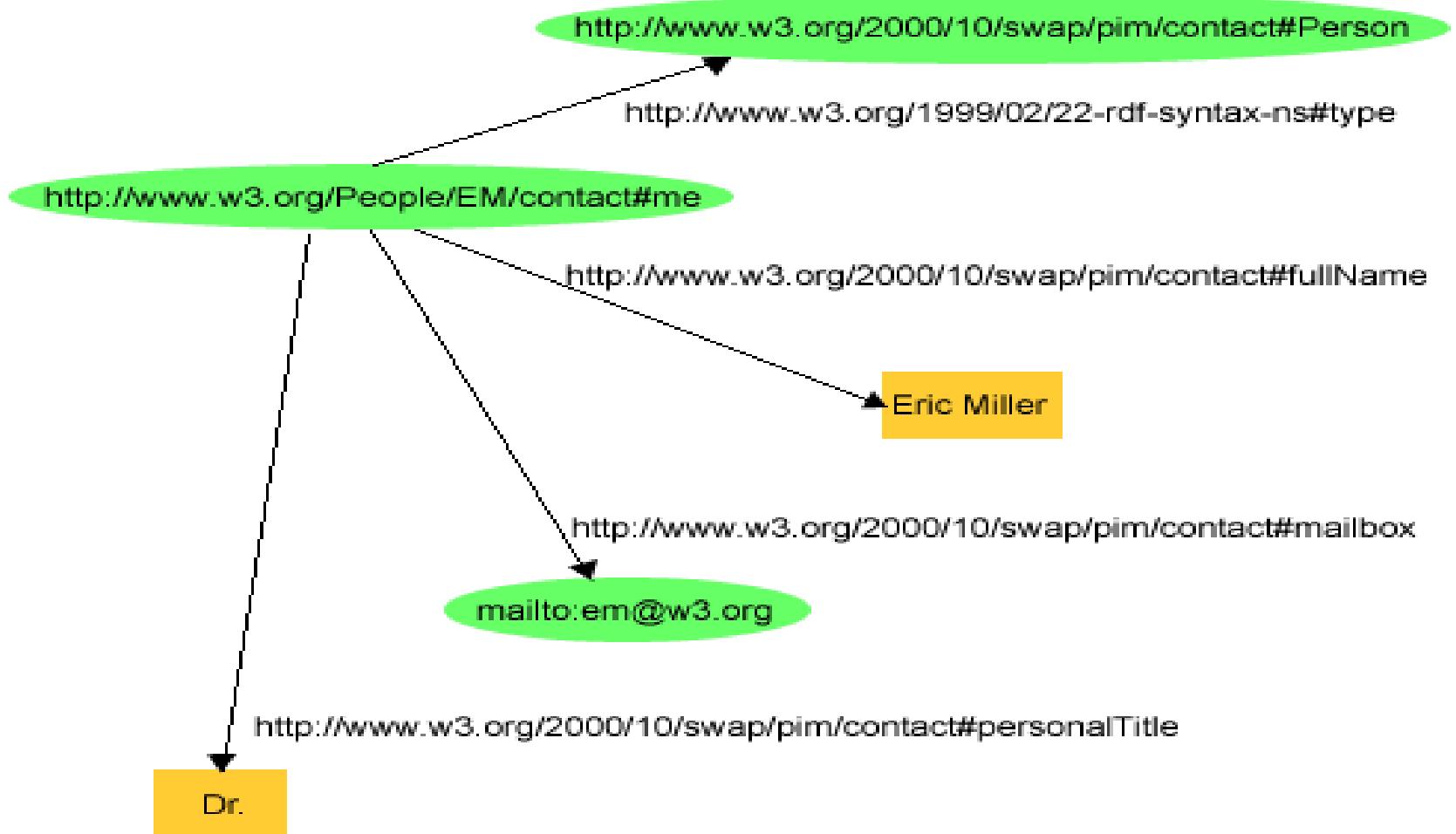
| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 22222 | Einstein | Physics | 95000 |
| 12121 | Wu | Finance | 90000 |
| 32343 | El Said | History | 60000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 98345 | Kim | Elec. Eng. | 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 |

(a) The *instructor* table

| <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|------------------|-----------------|---------------|
| Comp. Sci. | Taylor | 100000 |
| Biology | Watson | 90000 |
| Elec. Eng. | Taylor | 85000 |
| Music | Packard | 80000 |
| Finance | Painter | 120000 |
| History | Painter | 50000 |
| Physics | Watson | 70000 |

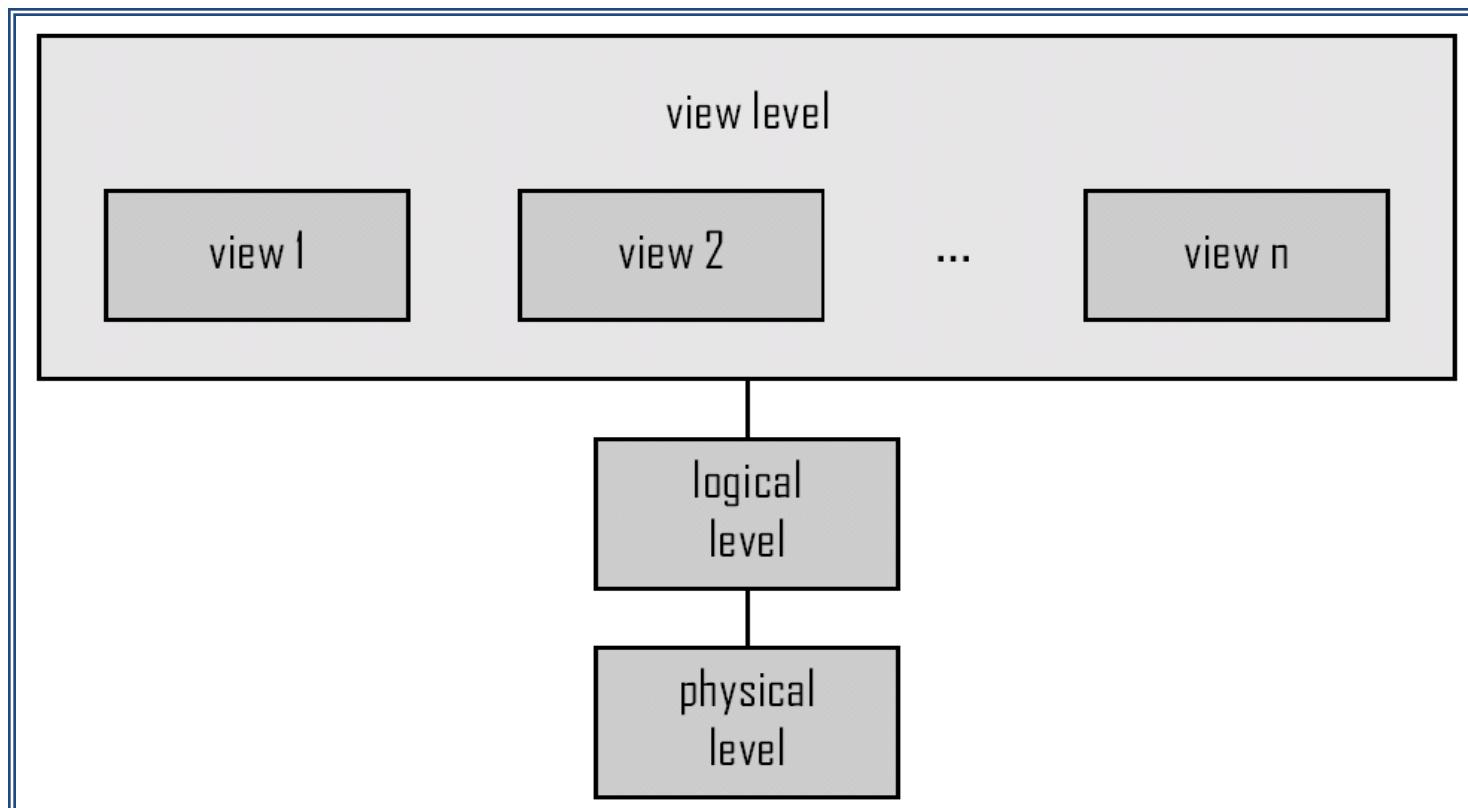
(b) The *department* table

RDF Model



Levels of Abstraction

- Databases provide users with an abstract view of data



3 Levels of Abstraction

- Physical or **Internal Level**
 - Lowest level of abstraction describes how data are actually stored
 - Describes complex low-level data structures in detail
- Logical or **Conceptual Level**
 - Describes what data are stored in the DB & what relationships exist among those data
 - Describes the entire DB in terms of relatively simpler structures
- View or **External Level**
 - Highest level of abstraction which describes only a part of the DB
 - User's view of the DB. This level describes part of the DB that is relevant to each user

Data Independence

- Applications insulated from how data is structured and stored
- Logical data independence: Protection from changes in logical structure of data
- Physical data independence: Protection from changes in physical structure of data

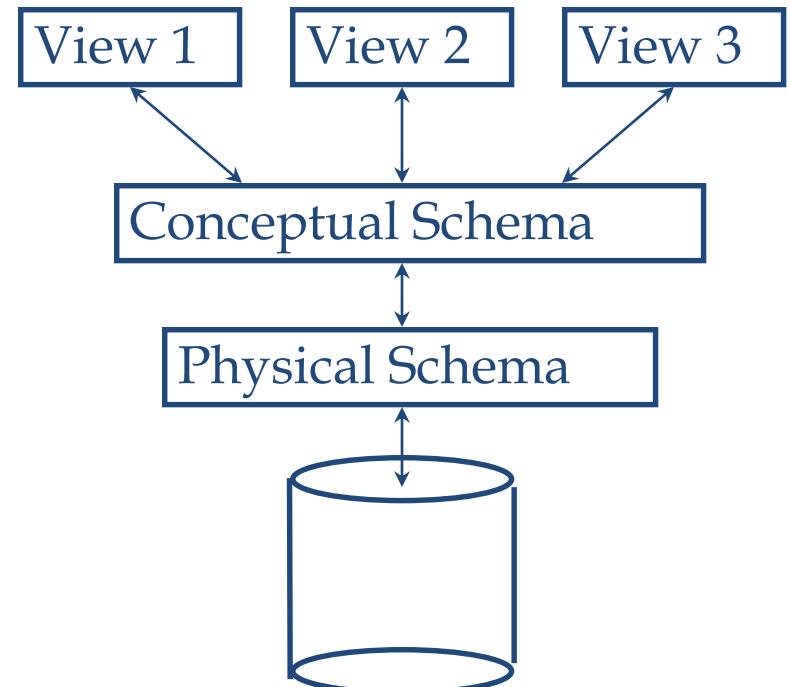
Levels of Abstraction

Many views, single conceptual (logical) schema and physical schema

Views describe how users see the data

Conceptual schema defines logical structure

Physical schema describes the files and indexes used



* *Schemas are defined using DDL; data is modified/queried using DML*

Instances and Schemas

- **Logical Schema** – the overall logical structure of the database
 - Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them
 - Analogous to type information of a variable in a program
- **Physical schema** – the overall physical structure of the database, indexes, space allocation, data compression, encryption, access paths

DBMS: Data access optimized and storage minimized
- **Instance** – the actual content of the database at a particular point in time
 - Analogous to the value of a variable

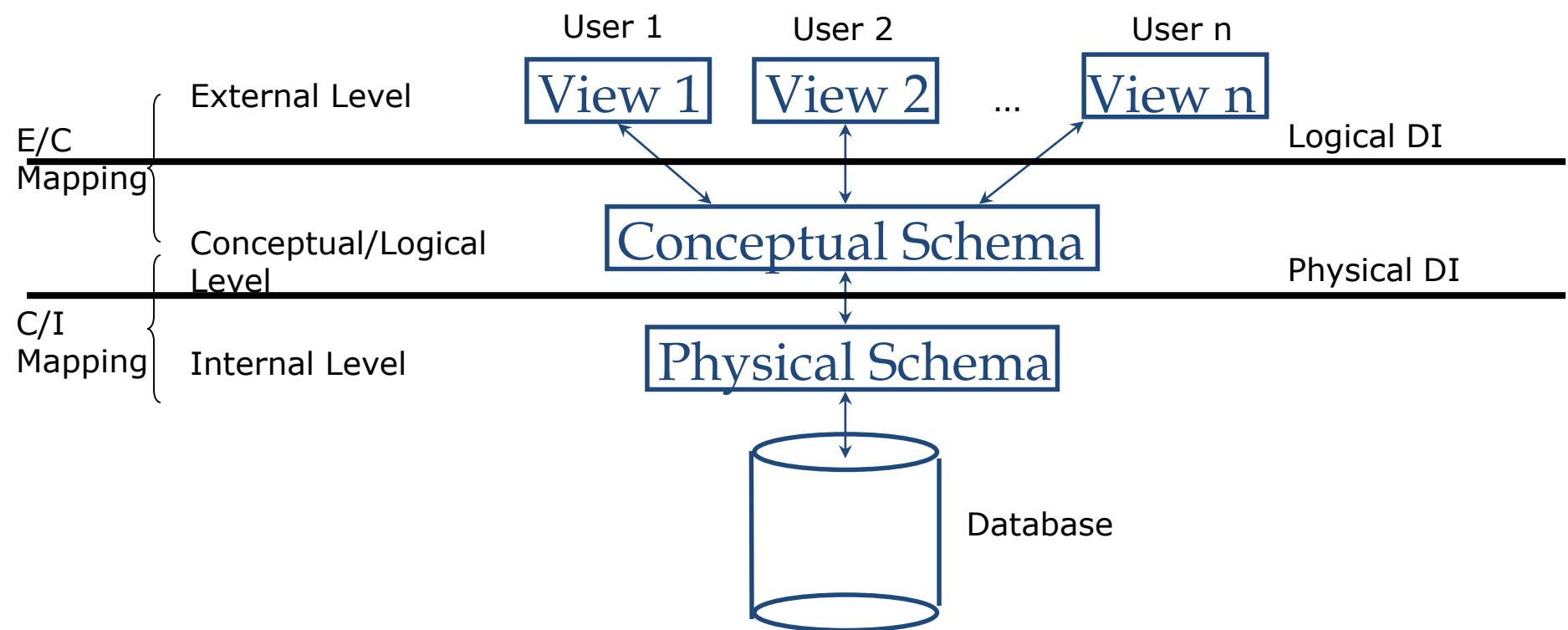
Example : University Database

- Conceptual schema:
 - *Students(sid: string, name: string, login: string, age: integer, gpa:real)*
 - *Courses(cid: string, cname:string, credits:integer)*
 - *Enrolled(sid:string, cid:string, grade:string)*
- Physical schema:
 - Relations stored as unordered files
 - Index on first column of Students
- External Schema (View):
 - *Course_info(cid:string,enrollment:integer)*

Physical Data Independence

- Applications insulated from how data is structured and stored
- Logical data independence: Protection from changes in logical structure of data
- Physical data independence: Protection from changes in physical structure of data
 - the ability to modify the physical schema without changing the logical schema
 - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

ANSI/SPARC 3-Tier Architecture



ANSI-SPARC Architecture

- Separates users' view from the way in which the data is arranged physically or logically
- It hides the physical storage details from users: Users should not have to deal with physical database storage details. They should be allowed to work with the data itself, without concern for how it is physically stored
- The database administrator should be able to change the database storage structures without affecting the users' views : From time to time changes to the structure of an organisation's data will be required.
- The internal structure of the database should be unaffected by changes to the physical aspects of the storage : For example, a changeover to a new disk.

Data Definition Language (DDL)

- Specification notation for defining the database schema

Example: **create table** *instructor* (

| | |
|------------------|-----------------------|
| <i>ID</i> | char(5) , |
| <i>name</i> | varchar(20) , |
| <i>dept_name</i> | varchar(20) , |
| <i>salary</i> | numeric(8,2)) |

- **DDL compiler** generates a set of table templates stored in a ***data dictionary***
- Data dictionary contains metadata (i.e., **data about data**)
 - Database schema
 - Integrity constraints
 - Primary key (ID uniquely identifies instructors)
 - Authorization
 - Who can access what

Data Manipulation Language (DML)

- Language for accessing and updating the data organized by the appropriate data model
 - DML also known as query language
- There are basically two types of data-manipulation language
 - **Procedural DML** -- require a user to specify what data are needed and how to get those data.
 - **Declarative DML** -- require a user to specify what data are needed without specifying how to get those data.
- Declarative DMLs are usually easier to learn and use than are procedural DMLs.
- Declarative DMLs are also referred to as non-procedural DMLs
- The portion of a DML that involves information retrieval is called a **query** language.

SQL Query Language

- SQL query language is nonprocedural. A query takes as input several tables (or only one) and always returns a single table.
- Example to find all instructors in Comp. Sci. dept

```
select name  
from instructor  
where dept_name = 'Comp. Sci.'
```

- To be able to compute complex functions SQL is usually embedded in some higher-level language
- Application programs generally access databases through one of
 - Language extensions to allow embedded SQL
 - Application program interface API (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database

Database Access from Application Program

- SQL does not support actions such as input from users, output to displays, or communication over the network.
- Such computations and actions must be written in a **host language**, such as C/C++, Java or Python, with embedded **SQL queries** that access the data in the database.
- **Application programs** -- are programs that are used to interact with the database in this fashion.

Database Design

- Logical Design – Deciding on the database schema.
Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database

Database Engine

- A database system is partitioned into modules that deal with each of the responsibilities of the overall system.
- The functional components of a database system can be divided into
 - The storage manager
 - The query processor component
 - The transaction management component

Storage Manager

- A program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
 - Interaction with the OS file manager
 - Efficient storing, retrieving and updating of data
- The storage manager components include:
 - Authorization and integrity manager
 - Transaction manager
 - File manager
 - Buffer manager

Storage Manager

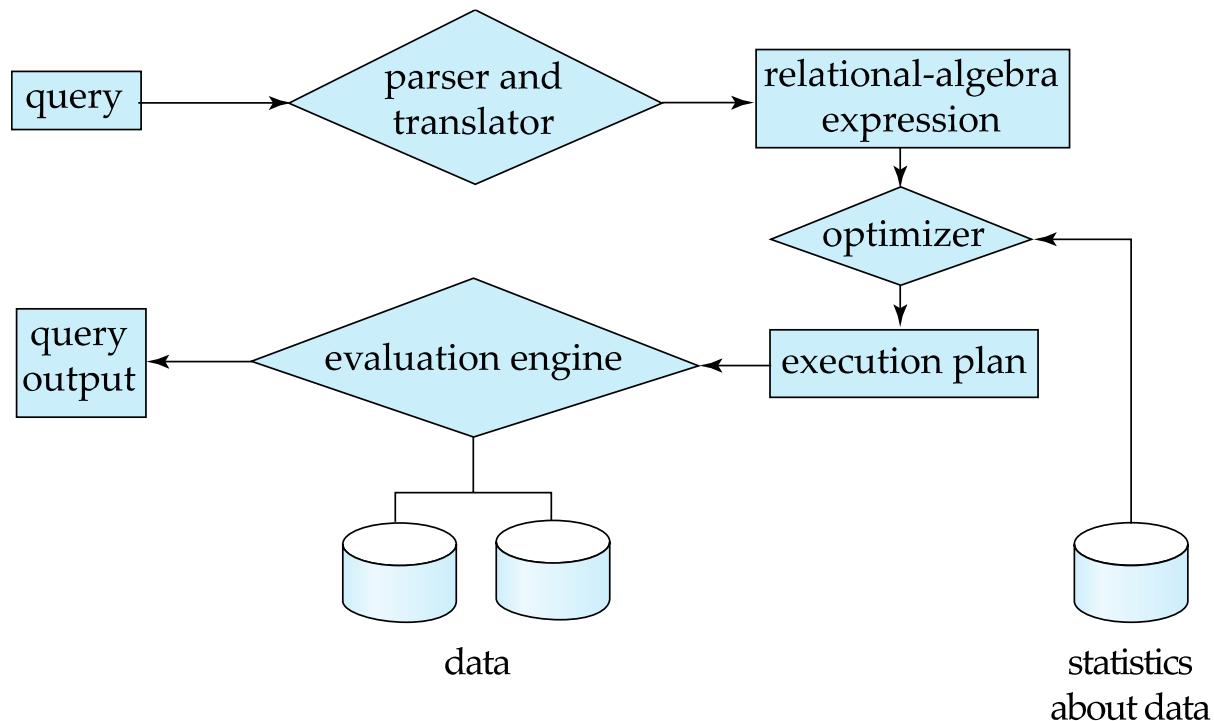
- The storage manager implements several data structures as part of the physical system implementation:
 - Data files -- store the database itself
 - Data dictionary -- stores metadata about the structure of the database, in particular the schema of the database.
 - Indices -- can provide fast access to data items. A database index provides pointers to those data items that hold a particular value.

Query Processor

- **DDL interpreter** -- interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler** -- translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
 - The DML compiler performs query optimization; that is, it picks the lowest cost evaluation plan from among the various alternatives.
- **Query evaluation engine** -- executes low-level instructions generated by the DML compiler.

Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Transaction Management

- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Concurrency Control

- Concurrent execution of user program is essential for good DBMS performance
 - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu working on several user programs concurrently
- Interleaving actions of different user programs can lead to inconsistencies
- DBMS ensures such problems don't arise: users can pretend they are using a single-user system

History of Database Systems

- 1950s and early 1960s:
 - Data processing using magnetic tapes for storage
 - Tapes provided only sequential access
 - Punched cards for input
- Late 1960s and 1970s:
 - Hard disks allowed direct access to data
 - Network and hierarchical data models in widespread use
 - Ted Codd defines the relational data model
 - Would win the ACM Turing Award for this work
 - IBM Research begins System R prototype
 - UC Berkeley (Michael Stonebraker) begins Ingres prototype
 - Oracle releases first commercial relational database
 - High-performance (for the era) transaction processing

History of Database Systems

- 1980s:
 - Research relational prototypes evolve into commercial systems
 - SQL becomes industrial standard
 - Parallel and distributed database systems
 - Wisconsin, IBM, Teradata
 - Object-oriented database systems
- 1990s:
 - Large decision support and data-mining applications
 - Large multi-terabyte data warehouses
 - Emergence of Web commerce

History of Database Systems

- 2000s
 - Big data storage systems
 - Google BigTable, Yahoo PNuts, Amazon,
 - “NoSQL” systems.
 - Big data analysis: beyond SQL
 - Map reduce
- 2010s
 - SQL reloaded
 - SQL front end to Map Reduce systems
 - Massively parallel database systems
 - Multi-core main-memory databases

Summary

- DBMS is used to maintain, query large volume of data
- Benefits of DBMS include recovery from system crashes, concurrent access, quick application development, data integrity and security
- Levels of abstraction give data independence
- A DBMS typically has a layered architecture

Relational Model

Outline

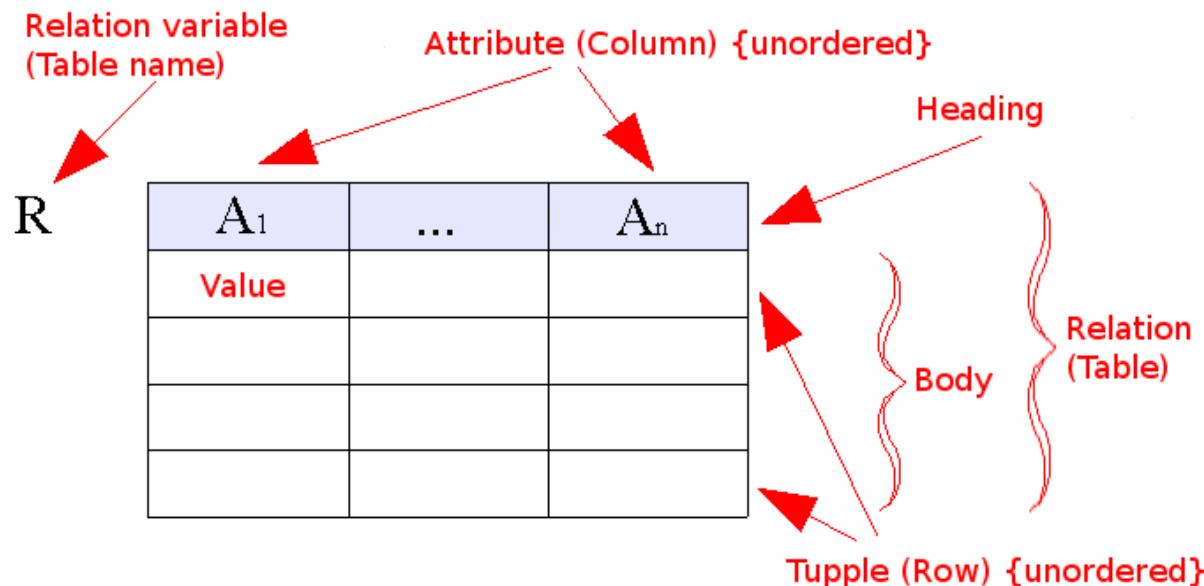
- Structure of Relational Databases
- Database Schema
- Keys
- Schema Diagrams
- Relational Query Languages
- The Relational Algebra

Relational Model Foundation

- Relational Model of data is based on the concept of RELATION
- A Relation is a Mathematical concept based on idea of SETS
- The strength of the relational approach to data management comes from the formal foundation provided by the theory of relations
- The model was first proposed by Dr. E.F. Codd of IBM in 1970

Relational Model (1970)

- Data as set of tables
- Having constraints and relationships
- Oracle, DB2, Sybase



Some Terms

Informal Name

- Table
- Row or Record
- Column or Field
- No. of Rows
- No. of Columns
- Unique Identifier
- Legal Values Set

Formal Name

- Relation
- Tuple
- Attribute
- Cardinality
- Degree or Arity
- Primary key
- Domain

Relational Database: Definitions

- Relational database: a set of relations
- Relation: made up of 2 parts:
 - Instance : a table, with rows and columns.
#Rows = cardinality, #fields = degree / arity.
 - Schema : specifies name of relation, plus name and type of each column.
 - E.G. Students(sid: string, name: string, login: string, age: integer, gpa: real).
- Can think of a relation as a set of rows or tuples (i.e., all rows are distinct).

Example of a *Instructor* Relation

The diagram illustrates a relation table for 'Instructor' with four columns: *ID*, *name*, *dept_name*, and *salary*. Three arrows point from the top right towards the table: one pointing to the first column labeled 'attributes (or columns)', another pointing to the second column labeled 'attributes (or columns)', and a third pointing to the fourth column labeled 'attributes (or columns)'. Three arrows point from the bottom right towards the table: one pointing to the first row labeled 'tuples (or rows)', another pointing to the second row labeled 'tuples (or rows)', and a third pointing to the third row labeled 'tuples (or rows)'.

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

Relation Schema and Instance

- A_1, A_2, \dots, A_n are *attributes*
- $R = (A_1, A_2, \dots, A_n)$ is a *relation schema*

Example:

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

- A relation instance r defined over schema R is denoted by $r(R)$.
- The current values a relation are specified by a table
- An element ***t*** of relation ***r*** is called a *tuple* and is represented by a *row* in a table

Attributes

- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value ***null*** is a member of every domain. Indicated that the value is “unknown”
- The null value causes complications in the definition of many operations

Null Values

- Comparisons with null values return the special truth value *null (unknown)*
- Three-valued logic using the truth value *null*:
 - **OR:** $(\text{null OR true}) = \text{true}$
 $(\text{null OR false}) = \text{null}$
 $(\text{null OR null}) = \text{null}$
 - **AND:** $(\text{true AND null}) = \text{null}$
 $(\text{false AND null}) = \text{false}$
 $(\text{null AND null}) = \text{null}$
 - **NOT:** $(\text{NOT null}) = \text{null}$

Attributes

- An **attribute** is a characteristic
 - Property, data element, field, attribute
 - Ex: Student (name, IDNo, course credit)
- **Domain**
- **Attributes can be**
 - **Atomic (simple) vs. Composite**
 - Ex: Student ID , Student name
 - Where student name may consists of (first, middle, last name)
 - **Single-valued vs. Multi-valued**
 - Ex. number of courses registered, names of registered courses
 - **Derived**
 - Ex: age of a student

Relations are Unordered

- Order of tuples is **irrelevant** (tuples may be stored in an arbitrary order)
- Example: *instructor* relation with unordered tuples

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 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 Schema

- Database schema -- is the logical structure of the database.
- Database instance -- is a snapshot of the data in the database at a given instant in time.
- Example:
 - schema: *instructor (ID, name, dept_name, salary)*
 - Instance:

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

Keys

- **Superkey (SK)**
 - Set of one or more attributes which taken collectively identifies uniquely an entity in entity set
 - Ex: Student (student name, IDNo, CPI, program, address)
 - Student entity set superkey can be (student name, IDNo)
 - Superkey may contain extraneous attributes
 - There can be more than one superkeys
- **Candidate Key (CK)**
 - A superkey for which no subset is a superkey
 - Ex: IDNo is a Candidate Key as it is minimal and uniquely identifies a student from Student Entity Set
 - There can be more than one candidate keys

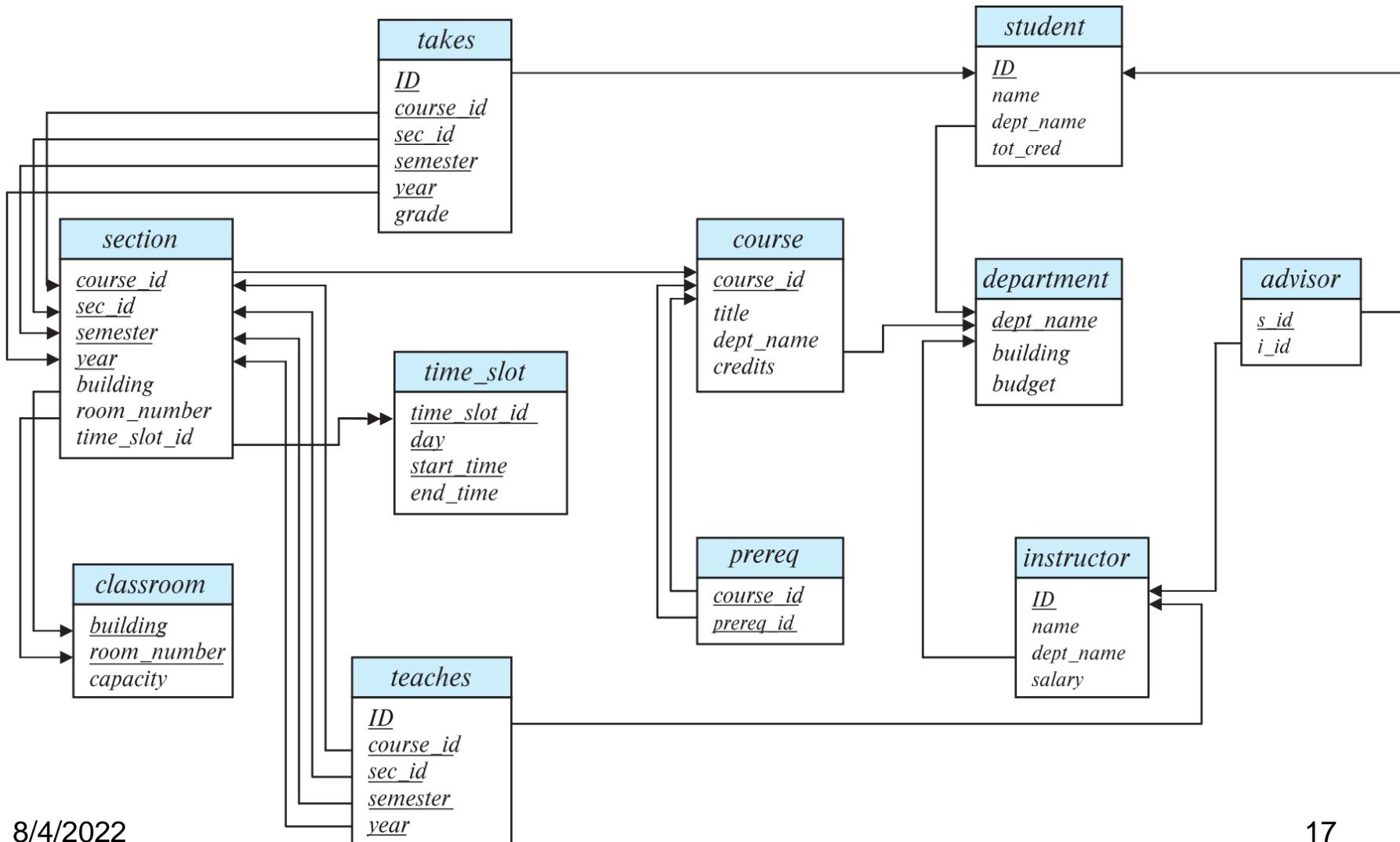
Keys

- **Primary Key (PK)**
 - Is a candidate key
 - One of the candidate key is chosen by database designer as a primary key
- **Example**
 - Telephone Book(STD Code, Telephone number, Name, Address)
 - Composite Key

Keys

- Let $K \subseteq R$
- 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: Value in one relation must appear in another
 - **Referencing** relation
 - **Referenced** relation
 - Example: *dept_name* in *instructor* is a foreign key from *instructor* referencing *department*

Schema Diagram for University Database



Schema of University Database

- *classroom(building, room number, capacity)*
- *department(dept name, building, budget)*
- *course(course id, title, dept name, credits)*
- *instructor(ID, name, dept name, salary)*
- *section(course id, sec id, semester, year, building, room number, time slot id)*
- *teaches(ID, course id, sec id, semester, year)*
- *student(ID, name, dept name, tot cred)*
- *takes(ID, course id, sec id, semester, year, grade)*
- *advisor(s ID, i ID)*
- *time slot(time slot id, day, start time, end time)*
- *Prereq (course id, prereq id)*

Relational Query Languages

- Procedural versus non-procedural, or declarative
- “Pure” languages:
 - Relational algebra
 - Tuple relational calculus
 - Domain relational calculus
- The above 3 pure languages are equivalent in computing power
- We will concentrate in this chapter on relational algebra
 - Consists of 6 basic operations

Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
 - select: σ
 - project: Π
 - union: \cup
 - set difference: –
 - Cartesian product: \times
 - rename: ρ

Relational Algebra Operators

Relational Algebra consists of several groups of operations

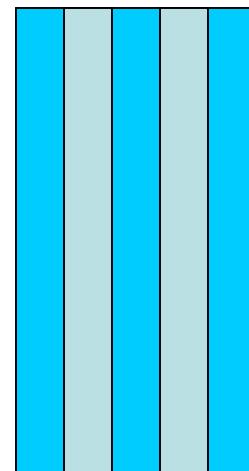
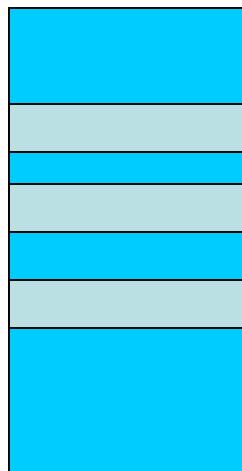
- Unary Relational Operations
 - SELECT (symbol: σ (sigma))
 - PROJECT (symbol: π (pi))
 - RENAME (symbol: ρ (rho))
- Relational Algebra Operations From Set Theory
 - UNION (\cup), INTERSECTION (\cap), DIFFERENCE (or MINUS, $-$)
 - CARTESIAN PRODUCT (\times)
- Binary Relational Operations
 - JOIN (several variations of JOIN exist)
 - DIVISION
- Additional Relational Operations
 - OUTER JOINS, OUTER UNION
 - AGGREGATE FUNCTIONS (These compute summary of information: for example, SUM, COUNT, AVG, MIN, MAX)

Relational Algebra

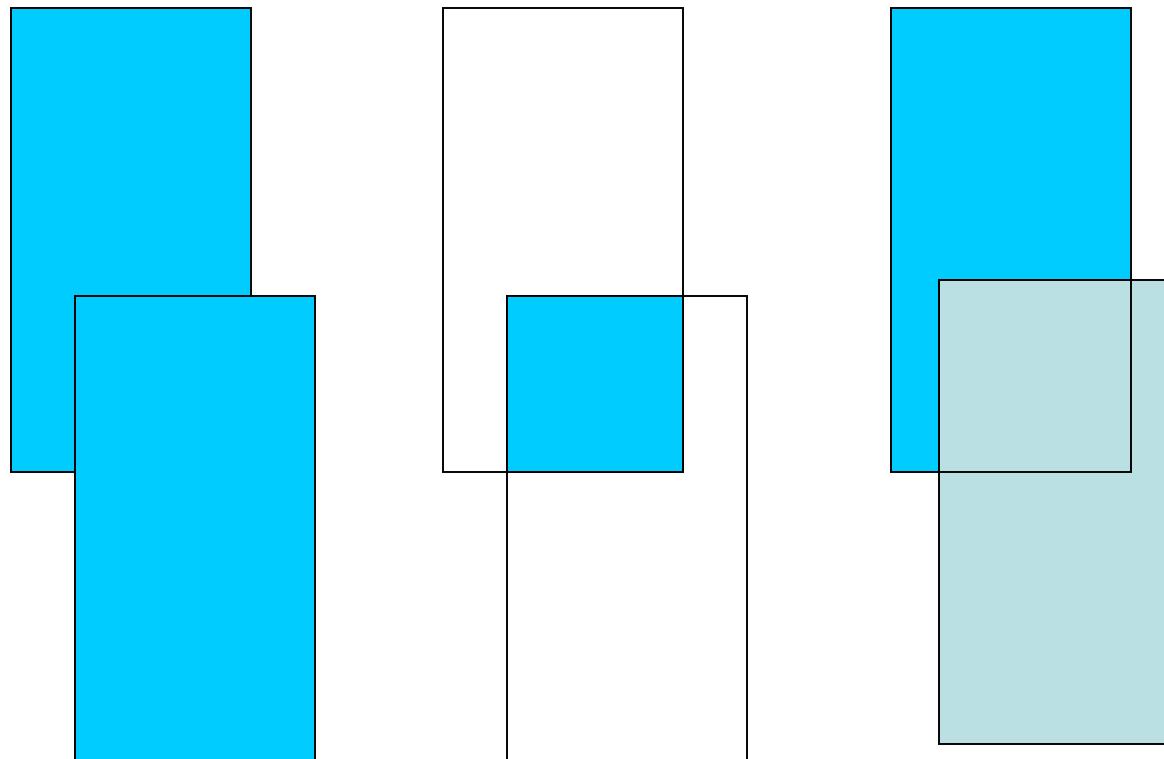
- Collection of operations on relations
- 8 operators + Rename operator
- Codd's 8 operators do not form a minimal set
- Some of them are not primitive
- Join, Intersect, & Divide can be defined in terms of other 5
- None of these 5 can be defined in terms of the remaining 4

Selection, Projection, Cartesian product, Union, and Set Difference.

Restrict & Project



Union, Intersection & Difference



Instructor

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

Select Operation

- The **select** operation selects tuples that satisfy a given predicate.
- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Example: select those tuples of the *instructor* relation where the instructor is in the “Physics” department.
 - Query

$$\sigma_{dept_name=\text{“Physics”}}(instructor)$$

- **Result**

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 22222 | Einstein | Physics | 95000 |
| 33456 | Gold | Physics | 87000 |

Select Operation

- We allow comparisons using
 $=, \neq, >, \geq, <, \leq$
in the selection predicate.
- We can combine several predicates into a larger predicate by using the connectives:
 \wedge (and), \vee (or), \neg (not)
- Example: Find the instructors in Physics with a salary greater than 90,000, we write:

$$\sigma_{dept_name=\text{"Physics"} \wedge salary > 90,000} (instructor)$$

- The select predicate may include comparisons between two attributes.
 - Example, find all departments whose name is the same as their building name:
 - $\sigma_{dept_name=building} (department)$

Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation:

$$\prod_{A_1, A_2, A_3 \dots A_k} (r)$$

where A_1, A_2, \dots, A_k are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets

for all relational algebra op. dups are removed

Project Operation Example

- Example: eliminate the *dept_name* attribute of *instructor*

$$\prod_{ID, name, salary} (\text{instructor})$$

- **Result:**

| <i>ID</i> | <i>name</i> | <i>salary</i> |
|-----------|-------------|---------------|
| 10101 | Srinivasan | 65000 |
| 12121 | Wu | 90000 |
| 15151 | Mozart | 40000 |
| 22222 | Einstein | 95000 |
| 32343 | El Said | 60000 |
| 33456 | Gold | 87000 |
| 45565 | Katz | 75000 |
| 58583 | Califieri | 62000 |
| 76543 | Singh | 80000 |
| 76766 | Crick | 72000 |
| 83821 | Brandt | 92000 |
| 98345 | Kim | 80000 |

Composition of Relational Operations

- The result of a relational-algebra operation is a relation and therefore relational-algebra operations can be composed together into a **relational-algebra expression**.
- *Find the names of all instructors in the Physics department.*

$$\prod_{name}(\sigma_{dept_name = "Physics"}(instructor))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

Cartesian-Product Operation

- The Cartesian-product operation (denoted by \times) allows us to combine information from any two relations.
- Example: the Cartesian product of the relations *instructor* and *teaches* is written as:
instructor* \times *teaches
- We construct a tuple of the result out of each possible pair of tuples: one from the *instructor* relation and one from the *teaches* relation
- Since the instructor *ID* appears in both relations we distinguish between these attributes by attaching to the attribute the name of the relation from which the attribute originally came.
 - *instructor.ID*
 - *teaches.ID*

The *instructor* x *teaches* table

| <i>instructor.ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> | <i>teaches.ID</i> | <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> |
|----------------------|-------------|------------------|---------------|-------------------|------------------|---------------|-----------------|-------------|
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-101 | 1 | Fall | 2017 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-315 | 1 | Spring | 2018 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-347 | 1 | Fall | 2017 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 12121 | FIN-201 | 1 | Spring | 2018 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 15151 | MU-199 | 1 | Spring | 2018 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 22222 | PHY-101 | 1 | Fall | 2017 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 12121 | Wu | Finance | 90000 | 10101 | CS-101 | 1 | Fall | 2017 |
| 12121 | Wu | Finance | 90000 | 10101 | CS-315 | 1 | Spring | 2018 |
| 12121 | Wu | Finance | 90000 | 10101 | CS-347 | 1 | Fall | 2017 |
| 12121 | Wu | Finance | 90000 | 12121 | FIN-201 | 1 | Spring | 2018 |
| 12121 | Wu | Finance | 90000 | 15151 | MU-199 | 1 | Spring | 2018 |
| 12121 | Wu | Finance | 90000 | 22222 | PHY-101 | 1 | Fall | 2017 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 15151 | Mozart | Music | 40000 | 10101 | CS-101 | 1 | Fall | 2017 |
| 15151 | Mozart | Music | 40000 | 10101 | CS-315 | 1 | Spring | 2018 |
| 15151 | Mozart | Music | 40000 | 10101 | CS-347 | 1 | Fall | 2017 |
| 15151 | Mozart | Music | 40000 | 12121 | FIN-201 | 1 | Spring | 2018 |
| 15151 | Mozart | Music | 40000 | 15151 | MU-199 | 1 | Spring | 2018 |
| 15151 | Mozart | Music | 40000 | 22222 | PHY-101 | 1 | Fall | 2017 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 22222 | Einstein | Physics | 95000 | 10101 | CS-101 | 1 | Fall | 2017 |
| 22222 | Einstein | Physics | 95000 | 10101 | CS-315 | 1 | Spring | 2018 |
| 22222 | Einstein | Physics | 95000 | 10101 | CS-347 | 1 | Fall | 2017 |
| 22222 | Einstein | Physics | 95000 | 12121 | FIN-201 | 1 | Spring | 2018 |
| 22222 | Einstein | Physics | 95000 | 15151 | MU-199 | 1 | Spring | 2018 |
| 22222 | Einstein | Physics | 95000 | 22222 | PHY-101 | 1 | Fall | 2017 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

Join Operation

- The Cartesian-Product
instructor X teaches

associates every tuple of instructor with every tuple of teaches.

- Most of the resulting rows have information about instructors who did NOT teach a particular course.
- To get only those tuples of “*instructor X teaches*” that pertain to instructors and the courses that they taught, we write:

$$\sigma_{instructor.id = teaches.id} (instructor \times teaches)$$

- We get only those tuples of “*instructor X teaches*” that pertain to instructors and the courses that they taught.

Join Operation (Cont.)

- $\sigma_{instructor.id = teaches.id} (instructor \times teaches)$

| <i>instructor.ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> | <i>teaches.ID</i> | <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> |
|----------------------|-------------|------------------|---------------|-------------------|------------------|---------------|-----------------|-------------|
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-101 | 1 | Fall | 2017 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-315 | 1 | Spring | 2018 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-347 | 1 | Fall | 2017 |
| 12121 | Wu | Finance | 90000 | 12121 | FIN-201 | 1 | Spring | 2018 |
| 15151 | Mozart | Music | 40000 | 15151 | MU-199 | 1 | Spring | 2018 |
| 22222 | Einstein | Physics | 95000 | 22222 | PHY-101 | 1 | Fall | 2017 |
| 32343 | El Said | History | 60000 | 32343 | HIS-351 | 1 | Spring | 2018 |
| 45565 | Katz | Comp. Sci. | 75000 | 45565 | CS-101 | 1 | Spring | 2018 |
| 45565 | Katz | Comp. Sci. | 75000 | 45565 | CS-319 | 1 | Spring | 2018 |
| 76766 | Crick | Biology | 72000 | 76766 | BIO-101 | 1 | Summer | 2017 |
| 76766 | Crick | Biology | 72000 | 76766 | BIO-301 | 1 | Summer | 2018 |
| 83821 | Brandt | Comp. Sci. | 92000 | 83821 | CS-190 | 1 | Spring | 2017 |
| 83821 | Brandt | Comp. Sci. | 92000 | 83821 | CS-190 | 2 | Spring | 2017 |
| 83821 | Brandt | Comp. Sci. | 92000 | 83821 | CS-319 | 2 | Spring | 2018 |
| 98345 | Kim | Elec. Eng. | 80000 | 98345 | EE-181 | 1 | Spring | 2017 |

Join Operation (Cont.)

- The **join** operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.
- Consider relations $r(R)$ and $s(S)$
- Let “theta” be a predicate on attributes in the schema R “union” S. The join operation $r \bowtie_{\theta} s$ is defined as follows:

$$r \bowtie_{\theta} s = \sigma_{\theta}(r \times s)$$

- Thus

$$\sigma_{instructor.id = teaches.id} (instructor \times teaches))$$

- Can equivalently be written as

$$instructor \bowtie Instructor.id = teaches.id \text{ teaches.}$$

Union Operation

- The union operation allows us to combine two relations
- Notation: $r \cup s$
- For $r \cup s$ to be valid.
 1. r, s must have the **same arity** (same number of attributes)
 2. The attribute domains must be **compatible** (example: 2nd column of r deals with the same type of values as does the 2nd column of s)
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\prod_{course_id} (\sigma_{semester="Fall"} \wedge year=2017^{(section)})$$

\cup

$$\prod_{course_id} (\sigma_{semester="Spring"} \wedge year=2018^{(section)})$$

Union Operation

- Result of:

$$\Pi_{course_id} (\sigma_{semester="Fall"} \wedge year=2017^{(section)})$$
$$\cup$$
$$\Pi_{course_id} (\sigma_{semester="Spring"} \wedge year=2018^{(section)})$$

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

Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: $r \cap s$
- Assume:
 - r, s have the *same arity*
 - attributes of r and s are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

$$\begin{aligned} & \prod_{course_id} (\sigma_{semester="Fall"} \wedge year=2017 \\ & (section)) \cap \\ & \prod_{course_id} (\sigma_{semester="Spring"} \wedge year=2018 \\ & (section)) \end{aligned}$$

Result

| course_id |
|-----------|
| CS-101 |

Set Difference Operation

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation $r - s$
- Set differences must be taken between **compatible** relations.
 - r and s must have the **same arity**
 - attribute domains of r and s must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{course_id} (\sigma_{semester="Fall" \wedge year=2017} (section)) -$$
$$\Pi_{course_id} (\sigma_{semester="Spring" \wedge year=2018} (section))$$

| |
|-----------|
| course_id |
| CS-347 |
| PHY-101 |

The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by \leftarrow and works like assignment in a programming language.
- Example: Find all instructor in the “Physics” and Music department.

$Physics \leftarrow \sigma_{dept_name = "Physics"}(instructor)$

$Music \leftarrow \sigma_{dept_name = "Music"}(instructor)$

$Physics \cup Music$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ , is provided for that purpose
- The expression:

$$\rho_x (E)$$

returns the result of expression E under the name x

- Another form of the rename operation:

$$\rho_{x(A_1, A_2, \dots A_n)} (E)$$

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
- Query 1

$$\sigma_{dept_name = "Physics"} \wedge salary > 90,000 \text{ (instructor)}$$

- Query 2

$$\sigma_{dept_name = "Physics"} (\sigma_{salary > 90.000} \text{ (instructor)})$$

- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department
- Query 1

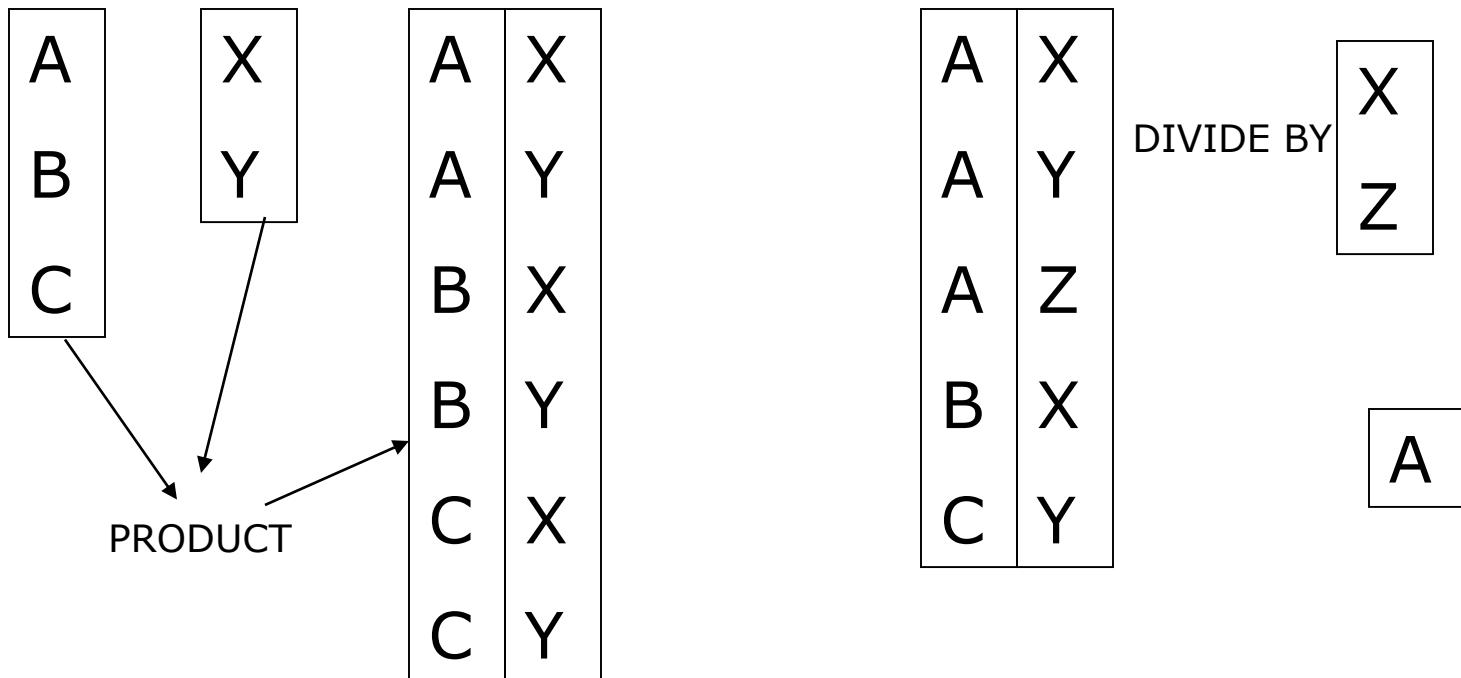
$$\sigma_{dept_name = "Physics"}(instructor \bowtie instructor.ID = teaches.ID \text{ teaches})$$

- Query 2

$$(\sigma_{dept_name = "Physics"}(instructor)) \bowtie instructor.ID = teaches.ID \text{ teaches}$$

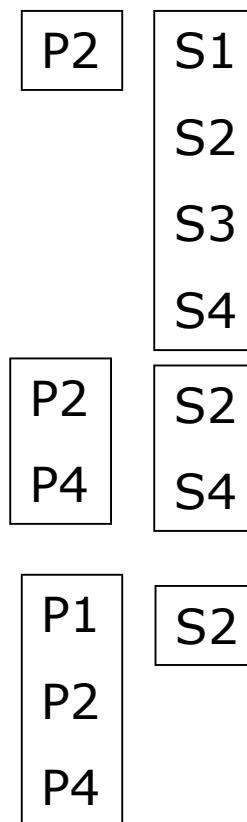
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Product & Divide



Division

| | |
|----|----|
| S1 | P1 |
| S1 | P2 |
| S2 | P3 |
| S2 | P4 |
| S2 | P1 |
| S2 | P2 |
| S3 | P2 |
| S4 | P2 |
| S4 | P4 |



P2 is in S1, S2, ...

P2 and P4 is in S2 and S4

P1,P2 and P4 is in S2



Chapter 3: Introduction to SQL

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

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

Outline

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

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999 (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.

SQL

- **Data Manipulation Language DML** -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the **Data Definition Language DDL** includes commands for specifying integrity constraints.
- **View definition** -- The DDL includes commands for defining views.
- *Transaction control* –includes commands for specifying the beginning and ending of transactions.
- *Embedded SQL and dynamic SQL* -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

Data Definition Language DDL

- *The schema for each relation.*
- *The type of values associated with each attribute.*
- *The Integrity constraints*
- The set of **indices to be maintained for each relation**.
- **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.

Create Table Construct

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

create table *r*

$$(A_1 D_1, A_2 D_2, \dots, A_n D_n, \\ (\text{integrity-constraint}_1), \\ \dots, \\ (\text{integrity-constraint}_k))$$

- *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

- Types of integrity constraints
 - **primary key** (A_1, \dots, A_n)
 - **foreign key** (A_m, \dots, A_n) **references** r
 - **not null**
- **SQL prevents any update to the database that violates an integrity constraint.**
- 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);
```

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*);

Table definitions

- **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** Delete is DML drop is DDL
 - `drop table r`
- **Alter**
 - `alter table r add A D`
 - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.
 - All existing tuples in the relation are assigned *null* as the value for the new attribute.
 - `alter table r drop A`
 - where *A* is the name of an attribute of relation *r*
 - Dropping of attributes not supported by many databases.

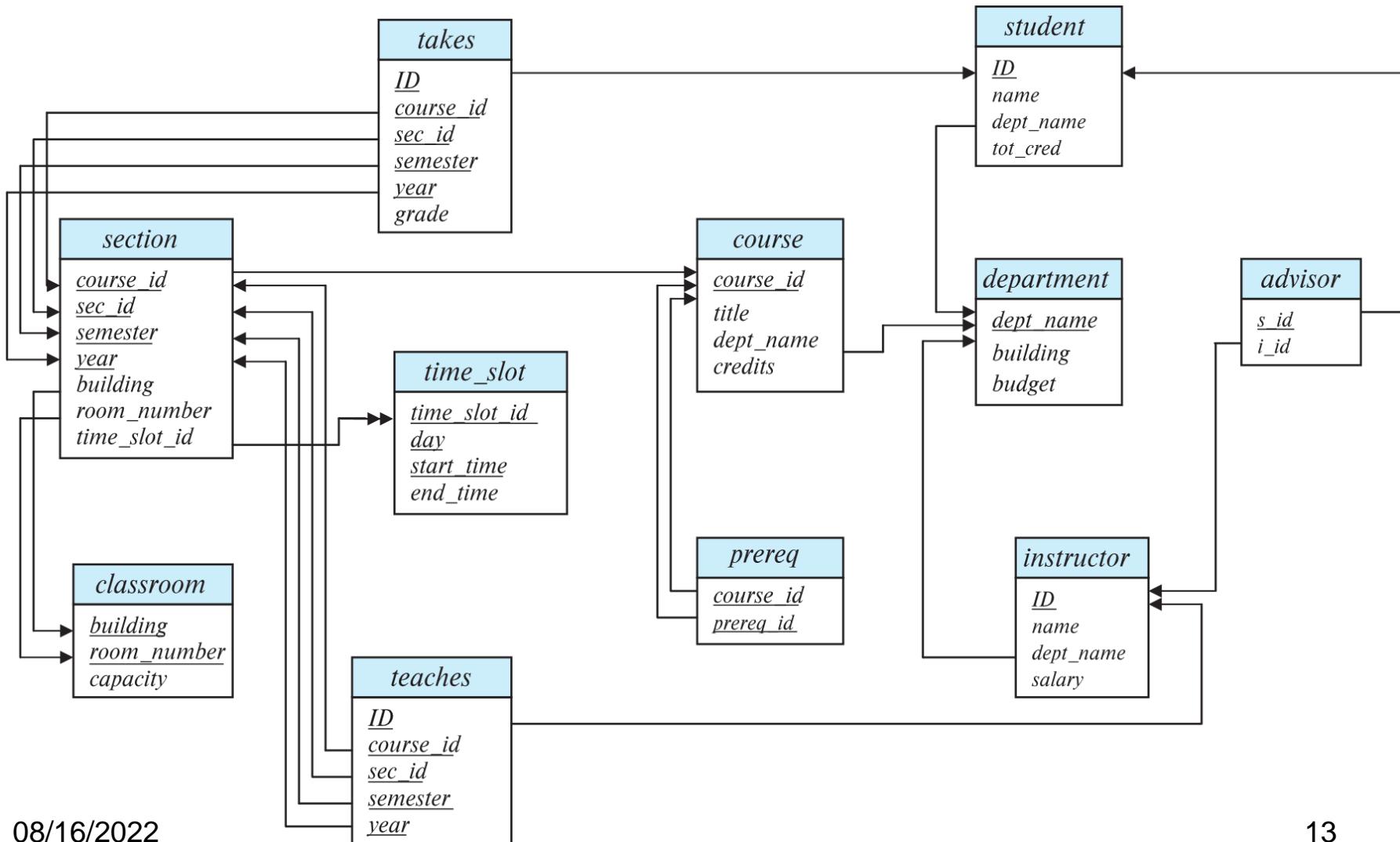
Basic Query Structure

- A typical SQL query has the form:

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

- A_i represents an attribute
- R_i represents a relation
- P is a predicate.
- The result of an SQL query is a relation.

Schema Diagram for University Database



Schema of University Database

- *classroom(building, room number, capacity)*
- *department(dept name, building, budget)*
- *course(course id, title, dept name, credits)*
- *instructor(ID, name, dept name, salary)*
- *section(course id, sec id, semester, year, building, room number, time slot id)*
- *teaches(ID, course id, sec id, semester, year)*
- *student(ID, name, dept name, tot cred)*
- *takes(ID, course id, sec id, semester, year, grade)*
- *advisor(s ID, i ID)*
- *time slot(time slot id, day, start time, end time)*
- *Prereq (course id, prereq id)*

The select Clause

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

```
select name  
from instructor
```

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
 - E.g., *Name* \equiv *NAME* \equiv *name*
 - Some people use upper case wherever we use bold font.

The select Clause

- 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
```

| <i>dept_name</i> |
|------------------|
| Comp. Sci. |
| Finance |
| Music |
| Physics |
| History |
| Physics |
| Comp. Sci. |
| History |
| Finance |
| Biology |
| Comp. Sci. |
| Elec. Eng. |

The select Clause

- 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

- 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.'
```

- SQL allows the use of the logical connectives **and**, **or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 70000

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

| name |
|--------|
| Katz |
| Brandt |

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).

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**
- 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'**

| <i>name</i> | <i>course_id</i> |
|-------------|------------------|
| Srinivasan | CS-101 |
| Srinivasan | CS-315 |
| Srinivasan | CS-347 |
| Wu | FIN-201 |
| Mozart | MU-199 |
| Einstein | PHY-101 |
| El Said | HIS-351 |
| Katz | CS-101 |
| Katz | CS-319 |
| Crick | BIO-101 |
| Crick | BIO-301 |
| Brandt | CS-190 |
| Brandt | CS-190 |
| Brandt | CS-319 |
| Kim | EE-181 |

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 \equiv *instructor T*

Self Join Example

- Relation *emp-super*

| <i>person</i> | <i>supervisor</i> |
|---------------|-------------------|
| Bob | Alice |
| Mary | Susan |
| Alice | David |
| David | Mary |

- Find the supervisor of “Bob”
- Find the supervisor of the supervisor of “Bob”
- Can you 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

- 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');
```

Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018
`(select course_id from section where sem = 'Fall' and year = 2017)`
`union`
`(select course_id from section where sem = 'Spring' and year = 2018)`
- Find courses that ran in Fall 2017 and in Spring 2018
`(select course_id from section where sem = 'Fall' and year = 2017)`
`intersect`
`(select course_id from section where sem = 'Spring' and year = 2018)`
- Find courses that ran in Fall 2017 but not in Spring 2018
`(select course_id from section where sem = 'Fall' and year = 2017)`
`except`
`(select course_id from section where sem = 'Spring' and year = 2018)`

Set Operations

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

Null Values

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

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

- The predicate **is not null** succeeds if the value on which it is applied is not null.

Null Values

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
 - Example: $5 < \text{null}$ or $\text{null} <> \text{null}$ or $\text{null} = \text{null}$
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
 - **and** : $(\text{true and unknown}) = \text{unknown}$,
 $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$
 - **or**: $(\text{unknown or true}) = \text{true}$,
 $(\text{unknown or false}) = \text{unknown}$
 $(\text{unknown or unknown}) = \text{unknown}$
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value
 - avg**: average value
 - min**: minimum value
 - max**: maximum value
 - sum**: sum of values
 - count**: number of values

Aggregate Functions Examples

- 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 2018 semester
  - ```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```
- 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;`

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

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

# Aggregation

- 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) as 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

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

as follows:

- **From clause:**  $r_i$  can be replaced by any valid subquery
- **Where clause:**  $P$  can be replaced with an expression of the form:  
 $B <\text{operation}> (\text{subquery})$   
 $B$  is an attribute and  $<\text{operation}>$  to be defined later.
- **Select clause:**  
 $A_i$  can be replaced by a subquery that generates a single value.

# **Set Membership**

# Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id in (select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id not in (select course_id
 from section
 where semester = 'Spring' and year= 2018);
```

# Set Membership

- Name all instructors whose name is neither “Mozart” nor Einstein”

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein')
```

- 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**

# Set Comparison – “some” Clause

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

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

- Same query using > **some** clause

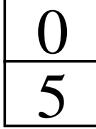
```
select name
from instructor
where salary > some (select salary
 from instructor
 where dept name = 'Biology');
```

# Definition of “some” Clause

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

$(5 < \mathbf{some}$    $) = \text{true}$       (read: 5 < some tuple in the relation)

$(5 < \mathbf{some}$    $) = \text{false}$

$(5 = \mathbf{some}$    $) = \text{true}$

$(5 \neq \mathbf{some}$    $) = \text{true}$  (since  $0 \neq 5$ )

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

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

# Set Comparison – “all” Clause

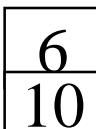
- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

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

# Definition of “all” Clause

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

(5 < all  ) = false

(5 < all  ) = true

(5 = all  ) = false

(5 ≠ all  ) = true (since  $5 \neq 4$  and  $5 \neq 6$ )

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

However, (= all)  $\neq$  in

# Test for Empty Relations

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

# Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester”

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
exists (select *
 from section as T
 where semester = 'Spring' and year= 2018
 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 that  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- 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 2017

```
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 = 2017);
```

# Subqueries in the From Clause

# Subqueries in the From Clause

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

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

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

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

# With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- 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

table dept\_total with rows

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

# Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- 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

# **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

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

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

- Problem: as we delete tuples from *instructor*, the average salary changes
- Solution used in SQL:
  1. First, compute **avg** (salary) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

# Insertion

- 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

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of \$18,000.

```
insert into instructor
```

```
 select ID, name, dept_name, 18000
 from student
 where dept_name = 'Music' and total_cred > 144;
```

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

Otherwise queries like

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

would cause problem

# Updates

- Give a 5% salary raise to all instructors

```
update instructor
 set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who **earn** less than 70000

```
update instructor
 set salary = salary * 1.05
where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
 set salary = salary * 1.05
where salary < (select avg (salary)
 from instructor);
```

# Updates

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



# Chapter 4 : Intermediate SQL

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use

# Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

# 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
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

# Schema of University Database

- *classroom(building, room number, capacity)*
- *department(dept name, building, budget)*
- *course(course id, title, dept name, credits)*
- *instructor(ID, name, dept name, salary)*
- *section(course id, sec id, semester, year, building, room number, time slot id)*
- *teaches(ID, course id, sec id, semester, year)*
- *student(ID, name, dept name, tot cred)*
- *takes(ID, course id, sec id, semester, year, grade)*
- *advisor(s ID, i ID)*
- *time slot(time slot id, day, start time, end time)*
- *Prereq (course id, prereq id)*

# Natural Join in SQL

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- List the names of students along with the course ID of the courses that they take
  - `select name, course_id  
from students, takes  
where student.ID = takes.ID;`
- Same query in SQL with “natural join” construct
  - `select name, course_id  
from student natural join takes;`

*student(ID, name, dept name, tot cred)*

*takes(ID, course id, sec id, semester, year, grade)*

# Natural Join in SQL

- The **from** clause can have multiple relations combined using natural join:

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

# Student Relation

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>tot_cred</i> |
|-----------|-------------|------------------|-----------------|
| 00128     | Zhang       | Comp. Sci.       | 102             |
| 12345     | Shankar     | Comp. Sci.       | 32              |
| 19991     | Brandt      | History          | 80              |
| 23121     | Chavez      | Finance          | 110             |
| 44553     | Peltier     | Physics          | 56              |
| 45678     | Levy        | Physics          | 46              |
| 54321     | Williams    | Comp. Sci.       | 54              |
| 55739     | Sanchez     | Music            | 38              |
| 70557     | Snow        | Physics          | 0               |
| 76543     | Brown       | Comp. Sci.       | 58              |
| 76653     | Aoi         | Elec. Eng.       | 60              |
| 98765     | Bourikas    | Elec. Eng.       | 98              |
| 98988     | Tanaka      | Biology          | 120             |

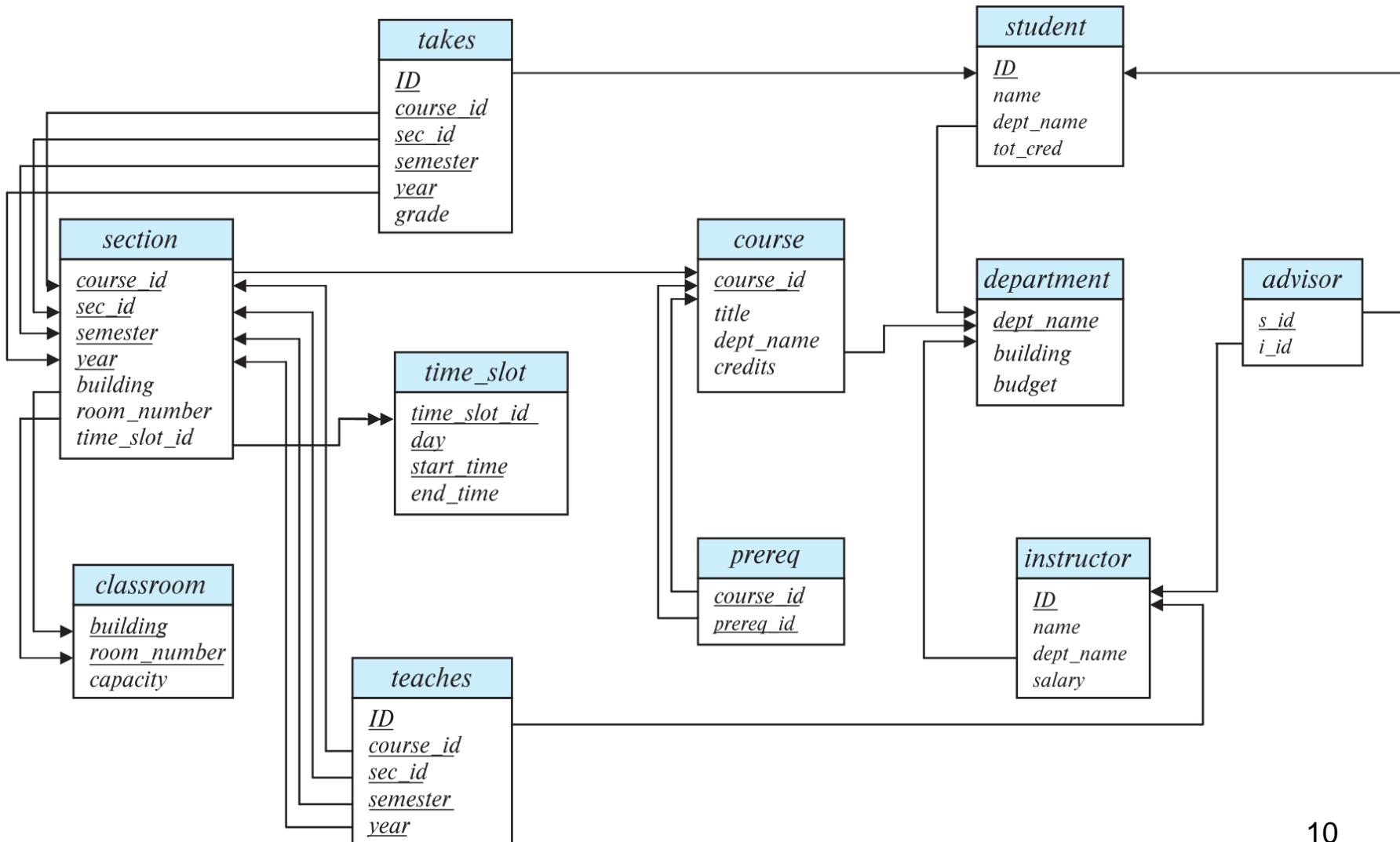
# Takes Relation

| <i>ID</i> | <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> | <i>grade</i> |
|-----------|------------------|---------------|-----------------|-------------|--------------|
| 00128     | CS-101           | 1             | Fall            | 2017        | A            |
| 00128     | CS-347           | 1             | Fall            | 2017        | A-           |
| 12345     | CS-101           | 1             | Fall            | 2017        | C            |
| 12345     | CS-190           | 2             | Spring          | 2017        | A            |
| 12345     | CS-315           | 1             | Spring          | 2018        | A            |
| 12345     | CS-347           | 1             | Fall            | 2017        | A            |
| 19991     | HIS-351          | 1             | Spring          | 2018        | B            |
| 23121     | FIN-201          | 1             | Spring          | 2018        | C+           |
| 44553     | PHY-101          | 1             | Fall            | 2017        | B-           |
| 45678     | CS-101           | 1             | Fall            | 2017        | F            |
| 45678     | CS-101           | 1             | Spring          | 2018        | B+           |
| 45678     | CS-319           | 1             | Spring          | 2018        | B            |
| 54321     | CS-101           | 1             | Fall            | 2017        | A-           |
| 54321     | CS-190           | 2             | Spring          | 2017        | B+           |
| 55739     | MU-199           | 1             | Spring          | 2018        | A-           |
| 76543     | CS-101           | 1             | Fall            | 2017        | A            |
| 76543     | CS-319           | 2             | Spring          | 2018        | A            |
| 76653     | EE-181           | 1             | Spring          | 2017        | C            |
| 98765     | CS-101           | 1             | Fall            | 2017        | C-           |
| 98765     | CS-315           | 1             | Spring          | 2018        | B            |
| 98988     | BIO-101          | 1             | Summer          | 2017        | A            |
| 98988     | BIO-301          | 1             | Summer          | 2018        | <i>null</i>  |

# ***student natural join takes***

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>tot_cred</i> | <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> | <i>grade</i> |
|-----------|-------------|------------------|-----------------|------------------|---------------|-----------------|-------------|--------------|
| 00128     | Zhang       | Comp. Sci.       | 102             | CS-101           | 1             | Fall            | 2017        | A            |
| 00128     | Zhang       | Comp. Sci.       | 102             | CS-347           | 1             | Fall            | 2017        | A-           |
| 12345     | Shankar     | Comp. Sci.       | 32              | CS-101           | 1             | Fall            | 2017        | C            |
| 12345     | Shankar     | Comp. Sci.       | 32              | CS-190           | 2             | Spring          | 2017        | A            |
| 12345     | Shankar     | Comp. Sci.       | 32              | CS-315           | 1             | Spring          | 2018        | A            |
| 12345     | Shankar     | Comp. Sci.       | 32              | CS-347           | 1             | Fall            | 2017        | A            |
| 19991     | Brandt      | History          | 80              | HIS-351          | 1             | Spring          | 2018        | B            |
| 23121     | Chavez      | Finance          | 110             | FIN-201          | 1             | Spring          | 2018        | C+           |
| 44553     | Peltier     | Physics          | 56              | PHY-101          | 1             | Fall            | 2017        | B-           |
| 45678     | Levy        | Physics          | 46              | CS-101           | 1             | Fall            | 2017        | F            |
| 45678     | Levy        | Physics          | 46              | CS-101           | 1             | Spring          | 2018        | B+           |
| 45678     | Levy        | Physics          | 46              | CS-319           | 1             | Spring          | 2018        | B            |
| 54321     | Williams    | Comp. Sci.       | 54              | CS-101           | 1             | Fall            | 2017        | A-           |
| 54321     | Williams    | Comp. Sci.       | 54              | CS-190           | 2             | Spring          | 2017        | B+           |
| 55739     | Sanchez     | Music            | 38              | MU-199           | 1             | Spring          | 2018        | A-           |
| 76543     | Brown       | Comp. Sci.       | 58              | CS-101           | 1             | Fall            | 2017        | A            |
| 76543     | Brown       | Comp. Sci.       | 58              | CS-319           | 2             | Spring          | 2018        | A            |
| 76653     | Aoi         | Elec. Eng.       | 60              | EE-181           | 1             | Spring          | 2017        | C            |
| 98765     | Bourikas    | Elec. Eng.       | 98              | CS-101           | 1             | Fall            | 2017        | C-           |
| 98765     | Bourikas    | Elec. Eng.       | 98              | CS-315           | 1             | Spring          | 2018        | B            |
| 98988     | Tanaka      | Biology          | 120             | BIO-101          | 1             | Summer          | 2017        | A            |
| 98988     | Tanaka      | Biology          | 120             | BIO-301          | 1             | Summer          | 2018        | <i>null</i>  |

# Schema Diagram for University Database



# Schema of University Database

- *classroom(building, room number, capacity)*
- *department(dept name, building, budget)*
- *course(course id, title, dept name, credits)*
- *instructor(ID, name, dept name, salary)*
- *section(course id, sec id, semester, year, building, room number, time slot id)*
- *teaches(ID, course id, sec id, semester, year)*
- *student(ID, name, dept name, tot cred)*
- *takes(ID, course id, sec id, semester, year, grade)*
- *advisor(s ID, i ID)*
- *time slot(time slot id, day, start time, end time)*
- *Prereq (course id, prereq id)*

# Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly
- Example -- List the names of students along with the titles of courses that they have taken
  - Correct version

```
select name, title
from student natural join takes, course
where takes.course_id = course.course_id;
```

- Incorrect version
  - This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
- **Course (course id, title, dept name, credits)**
- **Student (ID, name, dept name, tot cred)**
- **Takes (ID, course id, sec id, semester, year, grade)**

# Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
  - **left outer join**
  - **right outer join**
  - **full outer join**

# Outer Join Examples

- Relation ***course***

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

- Relation ***prereq***

| <i>course_id</i> | <i>prereq_id</i> |
|------------------|------------------|
| BIO-301          | BIO-101          |
| CS-190           | CS-101           |
| CS-347           | CS-101           |

- Observe that

course information is missing CS-437

prereq information is missing CS-315

# Left Outer Join

- **course natural left outer join prereq**

| <i>course_id</i> | <i>title</i> | <i>dept_name</i> | <i>credits</i> | <i>prereq_id</i> |
|------------------|--------------|------------------|----------------|------------------|
| BIO-301          | Genetics     | Biology          | 4              | BIO-101          |
| CS-190           | Game Design  | Comp. Sci.       | 4              | CS-101           |
| CS-315           | Robotics     | Comp. Sci.       | 3              | <i>null</i>      |

- In relational algebra: **course  $\bowtie$  prereq**

# Right Outer Join $\bowtie$

- $course \bowtie natural\ right\ outer\ join\ prereq$

| <i>course_id</i> | <i>title</i> | <i>dept_name</i> | <i>credits</i> | <i>prereq_id</i> |
|------------------|--------------|------------------|----------------|------------------|
| BIO-301          | Genetics     | Biology          | 4              | BIO-101          |
| CS-190           | Game Design  | Comp. Sci.       | 4              | CS-101           |
| CS-347           | <i>null</i>  | <i>null</i>      | <i>null</i>    | CS-101           |

- In relational algebra:  $course \bowtie prereq$

# 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    |

- In relational algebra: **course  prereq**

# Joined Types and Conditions

- **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.
- **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.

| <i>Join types</i>       |
|-------------------------|
| <b>inner join</b>       |
| <b>left outer join</b>  |
| <b>right outer join</b> |
| <b>full outer join</b>  |

| <i>Join conditions</i>                                          |
|-----------------------------------------------------------------|
| <b>natural</b>                                                  |
| <b>on &lt;predicate&gt;</b>                                     |
| <b>using (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>)</b> |

# Joined Relations – Examples

- 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    |

- 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    |

# Joined Relations – Examples

- **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    |

- What is the difference between the above, and a natural join?
- **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      |

*common cols will be eliminated for inner join*

# Joined Relations – Examples

- **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    |

- **course natural 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    |

# 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.

# View Definition and Use

- 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

- 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.

# Views Defined Using Other Views

- **create view *physics\_fall\_2017* 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 = '2017';**
- **create view *physics\_fall\_2017\_watson* as**  
**select course\_id, room\_number**  
**from *physics\_fall\_2017***  
**where building= 'Watson';**

# View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
 select course_id, room_number
 from physics_fall_2017
 where building= 'Watson'
```

- To: 

```
create view physics_fall_2017_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 = '2017')
 where building= 'Watson';
```

# 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

# Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized 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.
  - Storage
  - Stale results/ refresh / maintenance
  - Periodically or only when the view is accessed

# Update to a relation/database through 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 into the *instructor* relation
  - *Must have a value for salary.*
- Two approaches
  - Reject the insert
  - Insert 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/ or no departments in Taylor?
  - What if no department is in Taylor?
  - *Insert ('69987', 'White', null,null) into Instructor*
  - *Insert (null, 'Taylor', null) into Department*
  - These updates do not have the desired effect on instructor\_info as it still doesnot have a tuple **(69987, White, Taylor)**
  - There is no way to update Instructor and Department by using a null values in instructor and department to get desired update on instructor\_info

**Different database systems specify different conditions under which updatations through views are permitted.**

# And Some Not at All

- `create view history_instructors as  
 select *  
 from instructor  
 where dept_name= 'History';`
- What happens if we try to insert  
`('25566', 'Brown', 'Biology', 100000)`  
into *history\_instructors*?

*It will get inserted into **Instructor** but not visible in the **history\_instructors***

# View Updates in SQL

- 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*; that is, it does not have a not null constraint and is not part of a primary key
- The query does not have a **group by** or **having** clause.

# Transaction

- A **transaction** is a *unit* of program execution that **accesses and possibly updates various data items.**
- E.g., transaction to transfer \$50 from account A to account B:
  1. **read(A)**
  2.  $A := A - 50$
  3. **write(A)**
  4. **read(B)**
  5.  $B := B + 50$
  6. **write(B)**
- Two main issues to deal with:
  - Failures of various kinds, such as **hardware failures and system crashes**
  - **Concurrent execution of multiple transactions**
  - **ACID Properties**

*Atomicity, Consistency, Isolation, and Durability*

# Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a “unit” of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work**. The updates performed by the transaction become permanent in the database.
  - **Rollback work**. All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions

# Transfer of money

- Transaction to transfer \$50 from account A to account B:
  1. **read(A)**
  2.  $A := A - 50$
  3. **write(A)**
  4. **read(B)**
  5.  $B := B + 50$
  6. **write(B)**
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

# Transfer of money

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

# Transfer of money

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be).

T1

1. **read(A)**
2.  $A := A - 50$
3. **write(A)**
4. **read(B)**
5.  $B := B + 50$
6. **write(B)**

T2

read(A), read(B), print(A+B)

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

# Integrity

- Assurance that data always remains as intended
- Storage, retrieval, processing
- Integrity Constraints
  - Entity Integrity
  - Referential Integrity
  - Domain Integrity
  - User Defined Integrity

# 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

- **Create table**
- **Alter table** *tablename add constraint*
- Constraint is added to the relation only if it satisfies the constraint (at the time of executing this alter table command)

# Constraints on a Single Relation

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

# Not Null Constraints

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

*name* **varchar(20) not null**

*budget* **numeric(12,2) not null**

# Unique Constraints

- **unique** ( $A_1, A_2, \dots, A_m$ )
  - The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  form a candidate key or a superkey
  - No two tuples in the relation can be equal on all the ( $A_1, \dots, A_m$ )
  - Attributes in unique are permitted to be null (in contrast to primary keys) unless explicitly declared to be **non null**

# The check clause

- The **check (P)** clause specifies a predicate P that must be satisfied by every tuple in a relation.
- 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')))
```

Here **check** allows us to make domain constraint (**varchar**) more restrictive

# Check

- **create table** *department*  
*(dept name varchar (20),*  
*building varchar (15),*  
*budget numeric (12,2) check (budget > 0),*  
**primary key** (*dept name*));

## In create table statement:

- Put simple attribute value check along with the attribute
- Complex checks can be placed at the end of the create table statement

# 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.
  - Instructor (ID, name, dept\_name, salary)
  - Department ( dept\_name, building, budget)

# Referential Integrity

- Foreign keys *can be* specified as part of the SQL **create table** statement  
**foreign key (*dept\_name*) references department**
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.  
**foreign key (*dept\_name*) references department (*dept\_name*)**
- Instructor (ID, name, *dept\_name*, salary)
- Department ( dept\_name, building, budget)

# Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to enroll for courses

```
CREATE TABLE Enrolled
```

```
(sid CHAR(20), cid CHAR(20), grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid) REFERENCES Students)
```

Enrolled

| sid   | cid         | grade |
|-------|-------------|-------|
| 53666 | Carnatic101 | C     |
| 53666 | Reggae203   | B     |
| 53650 | Topology112 | A     |
| 53666 | History105  | B     |

Students

| sid   | name  | login      | age | gpa |
|-------|-------|------------|-----|-----|
| 53666 | Jones | jones@cs   | 18  | 3.4 |
| 53688 | Smith | smith@eecs | 18  | 3.2 |
| 53650 | Smith | smith@math | 19  | 3.8 |

# Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
 ...
 dept_name varchar(20),
 foreign key (dept_name) references department
 on delete cascade
 on update cascade,
 . . .)
```

- Instead of cascade we can use :
  - set null,
  - set default

Course( courseID, title,dept\_name, credits)

Department ( dept\_name, building, budget)

# Integrity Constraint Violation During Transactions

- Consider:

```
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

# Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

**check** (*time\_slot\_id* **in** (**select** *time\_slot\_id* **from** *time\_slot*))

The check condition states that the *time\_slot\_id* in each tuple in the *section* relation is actually the identifier of a time slot in the *time\_slot* relation.

- The condition has to be checked not only when a tuple is inserted or modified in *section*, but also when the relation *time\_slot* changes

**Section** (courseID, Sec\_ID, semester, year, building, room\_number, **timeslot\_ID**)

**Timeslot** (timeslot\_ID, day, start\_time, end\_time)

# Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot\_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:  
**create assertion <assertion-name> check (<predicate>);**

# 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

# Default value

```
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));
```

# Large-Object Types

- Large objects/ data items (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.
- *book review clob(10KB)*
- *image blob(10MB)*
- *movie blob(2GB)*

# User-Defined Types

*#define*

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

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

- Example:

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

# Domains

*user defined with constraint*

- **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.
- Example:

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

- **create domain DDollars as numeric(12,2) not null;**
- Can have constraints

# Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

**create index <name> on <relation-name> (attribute);**

# Index Creation Example

- **create index <name> on <relation-name> (attribute);**
- **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)**
- The query:

```
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

- **drop index <index-name>;**

# Authorization

- We may assign a user several forms of authorizations 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 tuples
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

RIUD

# Authorization

- 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 or view >
to <user list>
```

- Where <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role

```
grant select on department to Amita, Satoshi
```

# Authorization

- 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).
- **Update:** update any tuple in the relation  
`grant update (budget) on department to Amita;`

`budget` is attribute --> col

# 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 or view> from <user list>**

**Example:** **revoke select on student from U<sub>1</sub>, U<sub>2</sub>, U<sub>3</sub>**

- <privilege-list> may be **all** to revoke all privileges the revoker may hold.
- If <revoker-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.

**revoke update (budget) on department from Amita, Satoshi;**

# Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:  
**create a role <name>**
- Example:
  - **create role instructor**
- Once a role is created we can assign “users” to the role using:  
**grant <role> to <users>**

# Roles Example

- `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**;
  - Cascade is default
  - Restrict is required ex: deanship ends but instructor granted the privileges shouldn't be revoked as a result



# Chapter 5: Advanced SQL

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Accessing SQL From a Programming Language
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features



# Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.



# Accessing SQL from a Programming Language (Cont.)

There are two approaches to accessing SQL from a general-purpose programming language

- A general-purpose program -- can connect to and communicate with a database server using a collection of functions
- Embedded SQL -- provides a means by which a program can interact with a database server.
  - The SQL statements are translated at compile time into function calls.
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.



# JDBC



# JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle errors



# JDBC Code

```
public static void JDBCExample(String dbid, String userid, String passwd)
{
 try (Connection conn = DriverManager.getConnection(
 "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
 Statement stmt = conn.createStatement();
)
 {
 ... Do Actual Work
 }
 catch (SQLException sqle) {
 System.out.println("SQLException : " + sqle);
 }
}
```

**NOTE: Above syntax works with Java 7, and JDBC 4 onwards.**

**Resources opened in “try (...)” syntax (“try with resources”) are automatically closed at the end of the try block**



# JDBC Code for Older Versions of Java/JDBC

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
 try {
 Class.forName ("oracle.jdbc.driver.OracleDriver");
 Connection conn = DriverManager.getConnection(
 "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
 Statement stmt = conn.createStatement();
 ... Do Actual Work
 stmt.close();
 conn.close();
 }
 catch (SQLException sqle) {
 System.out.println("SQLException : " + sqle);
 }
}
```

**NOTE:** `Class.forName` is not required from JDBC 4 onwards. The try with resources syntax in prev slide is preferred for Java 7 onwards.



# JDBC Code (Cont.)

- Update to database

```
try {
 stmt.executeUpdate(
 "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
} catch (SQLException sqle)
{
 System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery(
 "select dept_name, avg (salary)
 from instructor
 group by dept_name");
while (rset.next()) {
 System.out.println(rset.getString("dept_name") + " " +
 rset.getFloat(2));
}
```



# JDBC SUBSECTIONS

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features
- Other Features
- Database Access from Python



# JDBC Code Details

- Getting result fields:
  - **rs.getString("dept\_name") and rs.getString(1) equivalent if dept\_name is the first argument of select result.**
- Dealing with Null values

```
int a = rs.getInt("a");
if (rs.wasNull()) System.out.println("Got null value");
```



# Prepared Statement

- ```
PreparedStatement pStmt = conn.prepareStatement(
        "insert into instructor values(?, ?, ?, ?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
 - NEVER create a query by concatenating strings
 - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept
name + " ', " ' balance + ')"
 - What if name is “D’Souza”?



SQL Injection

- Suppose query is constructed using
 - "select * from instructor where name = "" + name + """
- Suppose the user, instead of entering a name, enters:
 - X' or 'Y' = 'Y
- then the resulting statement becomes:
 - "select * from instructor where name = "" + "X' or 'Y' = 'Y" + """
 - which is:
 - select * from instructor where name = 'X' or 'Y' = 'Y'
 - User could have even used
 - X'; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
"select * from instructor where name = 'X\' or \'Y\' = \'Y'"
 - **Always use prepared statements, with user inputs as parameters**



Metadata Features

- ResultSet metadata
- E.g. after executing query to get a ResultSet rs:
 - ```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
 System.out.println(rsmd.getColumnName(i));
 System.out.println(rsmd.getColumnTypeName(i));
}
```
- How is this useful?



# Metadata (Cont)

- Database metadata

- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
// The value null indicates all Catalogs/Schemas.
// The value "" indicates current catalog/schema
// The value "%" has the same meaning as SQL like clause
```

```
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
```

```
while(rs.next()) {
```

```
 System.out.println(rs.getString("COLUMN_NAME"),
 rs.getString("TYPE_NAME"));
```

```
}
```

- And where is this useful?



# Metadata (Cont)

- Database metadata

- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern,
// and Table-Type
```

```
// Returns: One row for each table; row has a number of attributes
```

```
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
```

```
// The value null indicates all Catalogs/Schemas.
```

```
// The value "" indicates current catalog/schema
```

```
// The value "%" has the same meaning as SQL like clause
```

```
// The last attribute is an array of types of tables to return.
```

```
// TABLE means only regular tables
```

```
ResultSet rs = dbmd.getTables ("", "", "%", new String[] {"TABLES"});
```

```
while(rs.next()) {
```

```
 System.out.println(rs.getString("TABLE_NAME"));
```

```
}
```

- And where is this useful?



# Finding Primary Keys

- DatabaseMetaData dmd = connection.getMetaData();  
  
// Arguments below are: Catalog, Schema, and Table  
// The value “” for Catalog/Schema indicates current catalog/schema  
// The value null indicates all catalogs/schemas  
ResultSet rs = dmd.getPrimaryKeys("", "", tableName);  
  
while(rs.next()){  
 // KEY\_SEQ indicates the position of the attribute in  
 // the primary key, which is required if a primary key has multiple  
 // attributes  
 System.out.println(rs.getString("KEY\_SEQ"),  
 rs.getString("COLUMN\_NAME"));  
}



# Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - `conn.setAutoCommit(false);`
- Transactions must then be committed or rolled back explicitly
  - `conn.commit();`    or
  - `conn.rollback();`
- `conn.setAutoCommit(true)` turns on automatic commit.



# Other JDBC Features

- Calling functions and procedures
  - `CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");`
  - `CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)");`
- Handling large object types
  - `getBlob()` and `getClob()` that are similar to the `getString()` method, but return objects of type Blob and Clob, respectively
  - get data from these objects by `getBytes()`
  - associate an open stream with Java Blob or Clob object to update large objects
    - `blob.setBlob(int parameterIndex, InputStream inputStream).`



# JDBC Resources

- JDBC Basics Tutorial
  - <https://docs.oracle.com/javase/tutorial/jdbc/index.html>



# SQLJ

- JDBC is overly dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java

- ```
#sql iterator deptInfoIter ( String dept_name, int avgSal);
deptInfoIter iter = null;

#sql iter = { select dept_name, avg(salary) from instructor
              group by dept name };

while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}

iter.close();
```



ODBC



ODBC

- Open DataBase Connectivity (ODBC) standard
 - standard for application program to communicate with a database server.
 - application program interface (API) to
 - open a connection with a database,
 - send queries and updates,
 - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC



Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded SQL*.
- The basic form of these languages follows that of the System R embedding of SQL into PL/1.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement>;

Note: this varies by language:

- In some languages, like COBOL, the semicolon is replaced with END-EXEC
- In Java embedding uses # SQL { };



Embedded SQL (Cont.)

- Before executing any SQL statements, the program must first connect to the database. This is done using:

```
EXEC-SQL connect to server user user-name using password;
```

Here, *server* identifies the server to which a connection is to be established.

- Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., `:credit_amount`)
- Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

```
EXEC-SQL BEGIN DECLARE SECTION}
```

```
    int credit-amount ;
```

```
EXEC-SQL END DECLARE SECTION;
```



Embedded SQL (Cont.)

- To write an embedded SQL query, we use the
declare c cursor for <SQL query>
statement. The variable *c* is used to identify the query
- Example:
 - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable *credit_amount* in the host language
 - Specify the query in SQL as follows:

EXEC SQL

```
declare c cursor for
select ID, name
from student
where tot_cred > :credit_amount
```

END_EXEC



Embedded SQL (Cont.)

- The **open** statement for our example is as follows:

```
EXEC SQL open c ;
```

This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

```
EXEC SQL fetch c into :si, :sn END_EXEC
```

Repeated calls to fetch get successive tuples in the query result



Embedded SQL (Cont.)

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL close c ;
```

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.



Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- Can update tuples fetched by cursor by declaring that the cursor is for update

EXEC SQL

```
declare c cursor for
    select *
        from instructor
        where dept_name = 'Music'
        for update
```

- We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

```
update instructor
    set salary = salary + 1000
    where current of c
```



Functions and Procedures



Functions and Procedures

- Functions and procedures allow “business logic” to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
 - Most databases implement nonstandard versions of this syntax.



Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
    returns integer
begin
    declare d_count integer;
    select count (*) into d_count
        from instructor
        where instructor.dept_name = dept_name
    return d_count;
end
```

- The function *dept_count* can be used to find the department names and budget of all departments with more than 12 instructors.

```
select dept_name, budget
    from department
    where dept_count (dept_name) > 12
```



Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**

- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
```

```
    returns table (
```

```
        ID varchar(5),  
        name varchar(20),  
        dept_name varchar(20),  
        salary numeric(8,2))
```

```
    return table
```

```
        (select ID, name, dept_name, salary  
         from instructor  
         where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *  
from table (instructor_of ('Music'))
```



Language Constructs (Cont.)

- **For loop**
 - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;  
for r as  
    select budget from department  
    where dept_name = 'Music'  
do  
    set n = n + r.budget  
end for
```



External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
 - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL\can be executed by these functions.
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),  
                                out count integer)
```

language C

external name '/usr/avi/bin/dept_count_proc'

```
create function dept_count(dept_name varchar(20))
```

returns integer

language C

external name '/usr/avi/bin/dept_count'



Security with External Language Routines

- To deal with security problems, we can do one of the following:
 - Use **sandbox** techniques
 - That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code.
 - Run external language functions/procedures in a separate process, with no access to the database process' memory.
 - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space.



Triggers



Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
 - Specify the conditions under which the trigger is to be executed.
 - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
 - Syntax illustrated here may not work exactly on your database system; check the system manuals



Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
referencing new row as *nrow*
referencing old row as *orow*
for each row
when *nrow.grade* \neq 'F' **and** *nrow.grade* **is not null**
and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
begin atomic
 update *student*
 set *tot_cred*= *tot_cred* +
 (**select** *credits*
 from *course*
 where *course.course_id*= *nrow.course_id*)
 where *student.id* = *nrow.id*;
end;



Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
 - Use **for each statement** instead of **for each row**
 - Use **referencing old table** or **referencing new table** to refer to temporary tables (called **transition tables**) containing the affected rows
 - Can be more efficient when dealing with SQL statements that update a large number of rows



When Not To Use Triggers

- Triggers were used earlier for tasks such as
 - Maintaining summary data (e.g., total salary of each department)
 - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
 - Databases today provide built in materialized view facilities to maintain summary data
 - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
 - Define methods to update fields
 - Carry out actions as part of the update methods instead of through a trigger



When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
 - Loading data from a backup copy
 - Replicating updates at a remote site
 - Trigger execution can be disabled before such actions.
- Other risks with triggers:
 - Error leading to failure of critical transactions that set off the trigger
 - Cascading execution



Recursive Queries



Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
    select course_id, prereq_id
    from prereq
    union
    select rec_prereq.course_id, prereq.prereq_id,
    from rec_rereq, prereq
    where rec_prereq.prereq_id = prereq.course_id
)
select *
from rec_prereq;
```

This example view, *rec_prereq*, is called the *transitive closure* of the *prereq* relation



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 *prereq* with itself
 - This can give only a fixed number of levels of managers
 - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
 - Alternative: write a procedure to iterate as many times as required
 - See procedure *findAllPrereqs* in book



Example of Fixed-Point Computation

| <i>course_id</i> | <i>prereq_id</i> |
|------------------|------------------|
| BIO-301 | BIO-101 |
| BIO-399 | BIO-101 |
| CS-190 | CS-101 |
| CS-315 | CS-190 |
| CS-319 | CS-101 |
| CS-319 | CS-315 |
| CS-347 | CS-319 |

| <i>Iteration Number</i> | <i>Tuples in cI</i> |
|-------------------------|--|
| 0 | |
| 1 | (CS-319) |
| 2 | (CS-319), (CS-315), (CS-101) |
| 3 | (CS-319), (CS-315), (CS-101), (CS-190) |
| 4 | (CS-319), (CS-315), (CS-101), (CS-190) |
| 5 | done |



Advanced Aggregation Features



Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation
 $student_grades(ID, GPA)$
giving the grade-point average of each student
- Find the rank of each student.
- **select ID , rank() over (order by GPA desc) as s_rank**
from $student_grades$
- An extra **order by** clause is needed to get them in sorted order
select ID , rank() over (order by GPA desc) as s_rank
from $student_grades$
order by s_rank
- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
 - **dense_rank** does not leave gaps, so next dense rank would be 2



Ranking

- Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

```
select ID, (1 + (select count(*)  
                  from student_grades B  
                  where B.GPA > A.GPA)) as s_rank  
from student_grades A  
order by s_rank;
```



Ranking (Cont.)

- Ranking can be done within partition of the data.
- “Find the rank of students within each department.”

```
select ID, dept_name,
       rank () over (partition by dept_name order by GPA desc)
           as dept_rank
  from dept_grades
     order by dept_name, dept_rank;
```

- Multiple **rank** clauses can occur in a single **select** clause.
- Ranking is done *after* applying **group by** clause/aggregation
- Can be used to find top-n results
 - More general than the **limit n** clause supported by many databases, since it allows top-n within each partition



Ranking (Cont.)

- Other ranking functions:
 - **percent_rank** (within partition, if partitioning is done)
 - **cume_dist** (cumulative distribution)
 - fraction of tuples with preceding values
 - **row_number** (non-deterministic in presence of duplicates)
- SQL:1999 permits the user to specify **nulls first** or **nulls last**

```
select ID,  
      rank ( ) over (order by GPA desc nulls last) as s_rank  
from student_grades
```



Ranking (Cont.)

- For a given constant n , the ranking function $ntile(n)$ takes the tuples in each partition in the specified order, and divides them into n buckets with equal numbers of tuples.
- E.g.,

```
select ID, ntile(4) over (order by GPA desc) as quartile  
from student_grades;
```



Windowing

- Used to smooth out random variations.
- E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
- **Window specification** in SQL:
 - Given relation *sales(date, value)*
**select date, sum(value) over
(order by date between rows 1 preceding and 1 following)
from sales**



Windowing

- Examples of other window specifications:
 - **between rows unbounded preceding and current**
 - **rows unbounded preceding**
 - **range between 10 preceding and current row**
 - All rows with values between current row value –10 to current value
 - **range interval 10 day preceding**
 - Not including current row



Windowing (Cont.)

- Can do windowing within partitions
- E.g., Given a relation *transaction* (*account_number*, *date_time*, *value*), where value is positive for a deposit and negative for a withdrawal
 - “Find total balance of each account after each transaction on the account”

```
select account_number, date_time,  
    sum (value) over  
        (partition by account_number  
         order by date_time  
         rows unbounded preceding)  
    as balance  
from transaction  
order by account_number, date_time
```



OLAP



Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
 - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.
 - **Measure attributes**
 - measure some value
 - can be aggregated upon
 - e.g., the attribute *number* of the *sales* relation
 - **Dimension attributes**
 - define the dimensions on which measure attributes (or aggregates thereof) are viewed
 - e.g., attributes *item_name*, *color*, and *size* of the *sales* relation



Example sales relation

| item_name | color | clothes_size | quantity |
|-----------|--------|--------------|----------|
| skirt | dark | small | 2 |
| skirt | dark | medium | 5 |
| skirt | dark | large | 1 |
| skirt | pastel | small | 11 |
| skirt | pastel | medium | 9 |
| skirt | pastel | large | 15 |
| skirt | white | small | 2 |
| skirt | white | medium | 5 |
| skirt | white | large | 3 |
| dress | dark | small | 2 |
| dress | dark | medium | 6 |
| dress | dark | large | 12 |
| dress | pastel | small | 4 |
| dress | pastel | medium | 3 |
| dress | pastel | large | 3 |
| dress | white | small | 2 |
| dress | white | medium | 3 |
| dress | white | large | 0 |
| shirt | dark | small | 2 |
| shirt | dark | medium | 4 |
| ... | ... | ... | ... |



Cross Tabulation of sales by *item_name* and *color*

clothes_size all

| | | color | | |
|------------------|-------|-------|--------|-------|
| | | dark | pastel | white |
| <i>item_name</i> | skirt | 8 | 35 | 10 |
| | dress | 20 | 10 | 5 |
| | shirt | 14 | 7 | 28 |
| | pants | 20 | 2 | 5 |
| | total | 62 | 54 | 48 |
| | | total | | |
| | | 53 | 35 | 49 |
| | | 27 | | |
| | | 164 | | |

- The table above is an example of a **cross-tabulation (cross-tab)**, also referred to as a **pivot-table**.
 - Values for one of the dimension attributes form the row headers
 - Values for another dimension attribute form the column headers
 - Other dimension attributes are listed on top
 - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.



Data Cube

- A **data cube** is a multidimensional generalization of a cross-tab
- Can have n dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube

| | | item_name | | | | | clothes_size | | |
|-------|--|-----------|-------|-------|-------|-----|--------------|-------|--------|
| | | skirt | dress | shirt | pants | all | all | small | medium |
| | | 8 | 20 | 14 | 20 | 62 | 34 | 4 | 16 |
| color | | dark | 4 | 7 | 6 | 12 | 29 | | |
| | | pastel | 2 | 8 | 5 | 7 | 22 | | |
| | | white | 35 | 10 | 7 | 2 | 54 | 21 | 18 |
| | | all | 10 | 8 | 28 | 5 | 48 | 9 | 45 |
| | | 53 | 38 | 49 | 27 | 164 | 77 | 42 | all |



Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
 - Can drill down or roll up on a hierarchy

clothes_size: all

| | <i>category</i> | <i>item_name</i> | <i>color</i> | | | |
|------------|-----------------|------------------|--------------|--------|-------|-------|
| | | | dark | pastel | white | total |
| womenswear | skirt | 8 | 8 | 10 | 53 | 88 |
| | dress | 20 | 20 | 5 | 35 | |
| | subtotal | 28 | 28 | 15 | | |
| menswear | pants | 14 | 14 | 28 | 49 | 76 |
| | shirt | 20 | 20 | 5 | 27 | |
| | subtotal | 34 | 34 | 33 | | |
| total | | 62 | 62 | 48 | | 164 |



Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
 - We use the value **all** is used to represent aggregates.
 - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

| <i>item_name</i> | <i>color</i> | <i>clothes_size</i> | <i>quantity</i> |
|------------------|--------------|---------------------|-----------------|
| skirt | dark | all | 8 |
| skirt | pastel | all | 35 |
| skirt | white | all | 10 |
| skirt | all | all | 53 |
| dress | dark | all | 20 |
| dress | pastel | all | 10 |
| dress | white | all | 5 |
| dress | all | all | 35 |
| shirt | dark | all | 14 |
| shirt | pastel | all | 7 |
| shirt | White | all | 28 |
| shirt | all | all | 49 |
| pant | dark | all | 20 |
| pant | pastel | all | 2 |
| pant | white | all | 5 |
| pant | all | all | 27 |
| all | dark | all | 62 |
| all | pastel | all | 54 |
| all | white | all | 48 |
| all | all | all | 164 |



Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes
- Example relation for this section
 $sales(item_name, color, clothes_size, quantity)$
- E.g., consider the query

```
select item_name, color, size, sum(number)
from sales
group by cube(item_name, color, size)
```

This computes the union of eight different groupings of the *sales* relation:

```
{ (item_name, color, size), (item_name, color),
  (item_name, size),      (color, size),
  (item_name),           (color),
  (size),                () }
```

where () denotes an empty **group by** list.

- For each grouping, the result contains the null value for attributes not present in the grouping.



Online Analytical Processing Operations

- Relational representation of cross-tab that we saw earlier, but with *null* in place of **all**, can be computed by
 - ```
select item_name, color, sum(number)
 from sales
 group by cube(item_name, color)
```
- The function **grouping()** can be applied on an attribute
  - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

```
select item_name, color, size, sum(number),
 grouping(item_name) as item_name_flag,
 grouping(color) as color_flag,
 grouping(size) as size_flag,
 from sales
 group by cube(item_name, color, size)
```



# Online Analytical Processing Operations

- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**
  - E.g., replace *item\_name* in first query by  
**decode( grouping(item\_name), 1, 'all', item\_name)**



# Extended Aggregation (Cont.)

- The **rollup** construct generates union on every prefix of specified list of attributes
- E.g.,

```
select item_name, color, size, sum(number)
from sales
group by rollup(item_name, color, size)
```

- Generates union of four groupings:

```
{ (item_name, color, size), (item_name, color), (item_name), () }
```

- Rollup can be used to generate aggregates at multiple levels of a hierarchy.
- E.g., suppose table *itemcategory(item\_name, category)* gives the category of each item. Then

```
select category, item_name, sum(number)
from sales, itemcategory
where sales.item_name = itemcategory.item_name
group by rollup(category, item_name)
```

would give a hierarchical summary by *item\_name* and by *category*.



# Extended Aggregation (Cont.)

- Multiple rollups and cubes can be used in a single group by clause
  - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

```
select item_name, color, size, sum(number)
from sales
group by rollup(item_name), rollup(color, size)
```

generates the groupings

$$\begin{aligned} & \{item\_name, ()\} \times \{(color, size), (color), ()\} \\ &= \{ (item\_name, color, size), (item\_name, color), (item\_name), \\ & \quad (color, size), (color), () \} \end{aligned}$$



# Online Analytical Processing Operations

- **Pivoting:** changing the dimensions used in a cross-tab is called
- **Slicing:** creating a cross-tab for fixed values only
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup:** moving from finer-granularity data to a coarser granularity
- **Drill down:** The opposite operation - that of moving from coarser-granularity data to finer-granularity data



# OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems.
- OLAP implementations using only relational database features are called **relational OLAP (ROLAP)** systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.



# OLAP Implementation (Cont.)

- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
  - Space and time requirements for doing so can be very high
    - $2^n$  combinations of **group by**
  - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on  $(item\_name, color)$  from an aggregate on  $(item\_name, color, size)$ 
      - For all but a few “non-decomposable” aggregates such as *median*
      - is cheaper than computing it from scratch
- Several optimizations available for computing multiple aggregates
  - Can compute aggregate on  $(item\_name, color)$  from an aggregate on  $(item\_name, color, size)$
  - Can compute aggregates on  $(item\_name, color, size)$ ,  $(item\_name, color)$  and  $(item\_name)$  using a single sorting of the base data



# End of Chapter 5